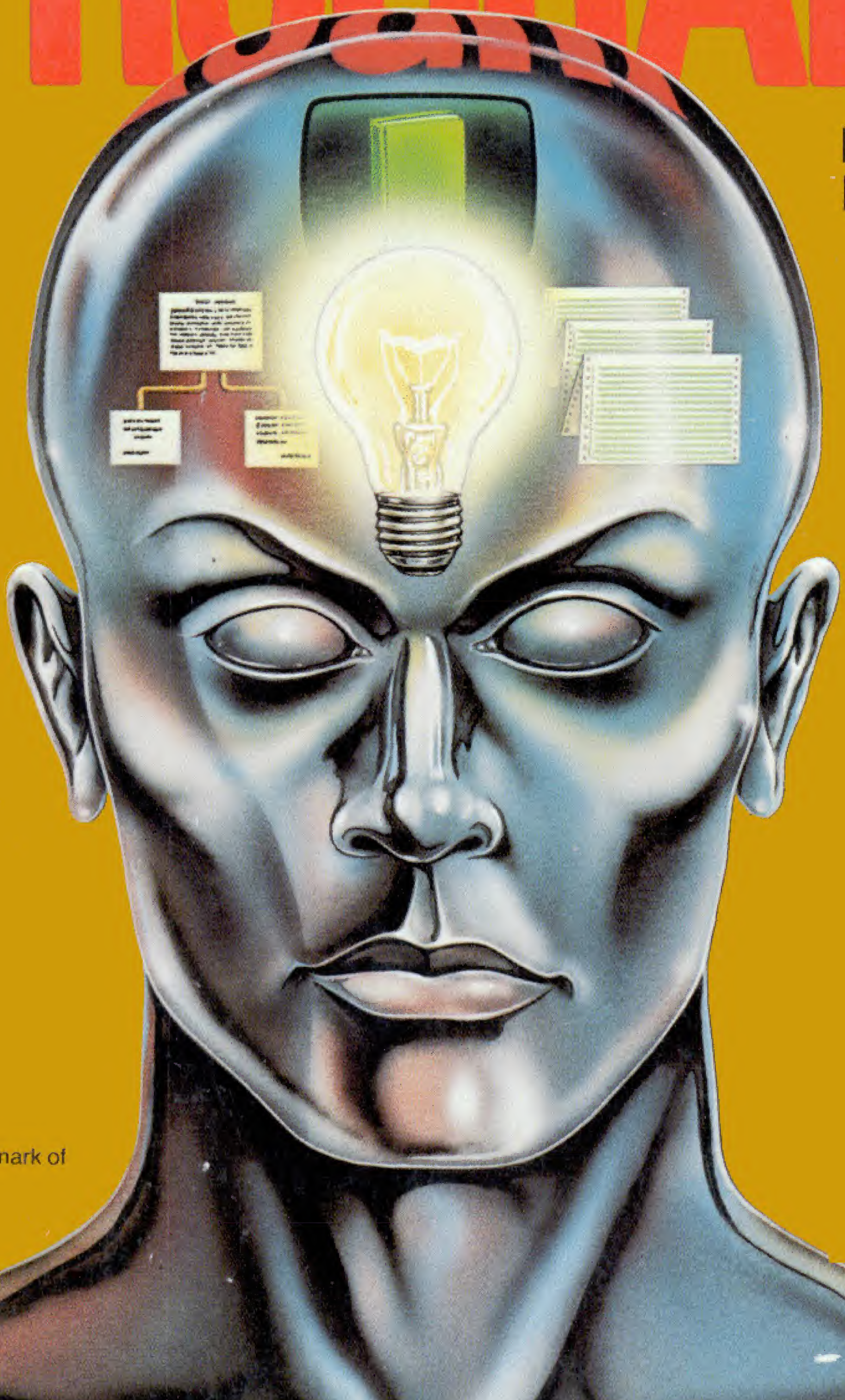
from DATAMOST, INC.

$14.95

REWARD BOOKS

HOW TO WRITE A TRS-80* PROGRAM

by Ed Faulk



*TRS-80 is a trademark of Tandy Corp.

HOW TO WRITE A COMPUTER PROGRAM

Volume I
TRS-80 EDITION
by Ed Faulk

illustrations by
Jeannine Likins

Published by Datamost, Inc.
9748 Cozycroft Avenue
Chatsworth, California 91311

Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia
Toll free (800) 336-0338
ISBN 08359-2992-2

TABLE OF CONTENTS

Introduction

During many years teaching computer programming languages, I was often amazed at the number of times students would ask the question "But, how do I actually write a program?" They were smart enough to realize that knowledge of a given computer programming language was not the same as being a programmer in that language.

In this book, I will provide some simple techniques for program development that have proven successful over the years, both for myself and others. The methods that we will be discussing have been called by various names over the last 10 years or so; sometimes the methods are called "modular programming," sometimes "structured programming," and sometimes "top-down devlopment." In any event, this technique is a mechanical attempt to define the sort of functions that the "top-notch" programmers use (sometimes unconsciously) so that all programmers can approach that level of proficiency.

While I won't guarantee that this book will turn you into a "super programmer," I will guarantee that these techniques can make you a better programmer if you will apply them diligently. The hard part, of course, is to believe that these methods will work for you! To prove that they will, we shall develop a program during the course of this book. The program is not a trivial one, and in fact is one that might be useful to you. This development will actually serve several purposes: (1) it will provide a simple set of examples for the topics that are being discussed; (2) it will prove that the techniques work, and (3) it will add to your library of subroutines.

As an overview to the book, let's take a look at the method by which I developed the chapters (functional subroutines of the book). There is an implied sequence in Sections 1 through 3, but Sections 4 through 6 really occur simultaneously upon the completion of Section 3. That is, you will get the idea, develop it and actually plan the program one step at a time. Once you begin coding the program you should also be testing and documenting as you go along. In a book, however, this is somewhat difficult to express, and so I have chosen to explain this here in the introduction to prevent later confusion. Those who decide to skip reading this may be in for a surprise when they get to Chapter 8, but who cares? They can always come back and read the introduction at a later date!

But, back to the overview, Chapters 1 through 3 provide the genesis, if you will, the foundation. Here we emphasize how impor-

tant it is to have an idea of what we want to do. Rarely does one sit down to write a program without knowing what the program is to do! In Section 1 we talk about the idea, how to get ideas, what an idea for a progam is, and how to embellish the idea. Section 2 carries the discussion to the next level, the planning of the program and the expansion of the idea, along with the evaluation of its suitability for a computer. Chapters 6 and 7 discuss two different ways of preparing a program for coding.

In Section 4 we discuss coding the program, or the implementation of the idea. Section 5 takes a look at how to go about testing the program while Section 6 lays out some guidelines for program documentation. In all, this book may be considered a guideline to program development. Obviously, it is not exhaustive in its treatment of the various topics, since each of the chapters has itself been the object of one or more books.

At the end of the book you will find a reference list of various books that may be helpful to you in the development of your own programs. I have read all of the books that are referenced and have found them to be quite useful to me.

One last idea before we jump into the book itself. Since this is a "how to" book, it assumes only a minimal knowledge of the computer that you are using, and a passing familiarity with BASIC. If you are not familiar with the computer, please read the manufacturers literature that comes with the computer. If you are not familiar with BASIC, don't worry about it too much. As we go along, the program examples and descriptions should explain the components of BASIC that we are using. It would still be a good idea to find one of the basic BASIC (no pun intended) books for your own study either before or after you read this book.

There's often a number of people that actually go into writing a book and this book is no exception. At the risk of forgetting some (here, not in my heart) I'd like to thank Dave Gordon who had faith in me, Ed Tischofer who wrote the Applesoft version of the CHEKBOOK program, Ron Markel who assisted in the IBM 5150 version of CHEKBOOK and Jeannine Likins whose marvelous illustrations grace the pages of this book. All of these people gave generously of their time and talents so that you could be reading this book. Also, since Murphy's law applies to books as well, I wish to state that I am solely responsible for all errors, all those who contributed are blameless.

Dedicated to Sandy, Don and Teri

Section 1 — The Idea

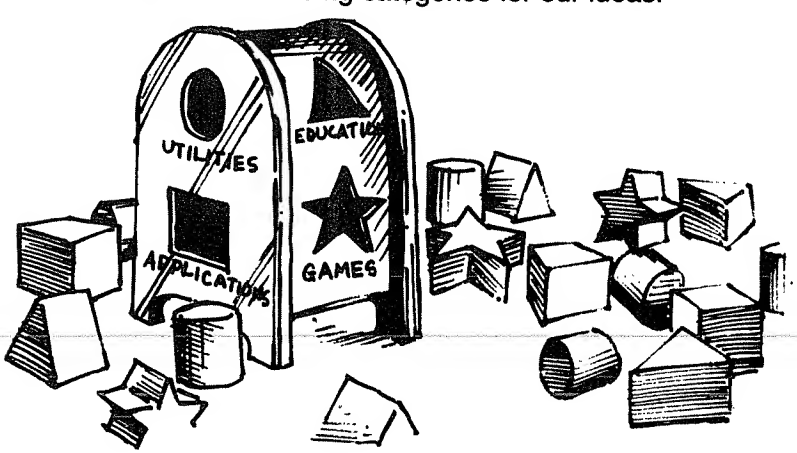
In this section we'll take a look at several different thoughts related to the development of program ideas, but not the programs themselves. We will be looking at the different ways to get ideas, the different categories of programs that these ideas will develop into, and ways to combine ideas for more effective programs. This section will provide the basis for the rest of the book by providing a foundation on which to build your development techniques.

CHAPTER 1

Idea Categories

As we think about the types of programs that have been written, it is easy to define some simple "pigeon-holes" into which they all fit. It is these "pigeon-holes" that form the various categories for our ideas. The reason for starting this book this way is to immediately begin orienting your thinking into the proper channels.

Programming has been called an art. But it is an art with scientific overtones. Although there is a sense of the artistic associated with really good programs, there is an underlying discipline that allows everyone to construct good working programs. It is this structure that we will be learning more about, and our look at ideas will reveal the reasons for building categories for our ideas.



Let's start with the simple categories. First, we have game programs. Games come in two basic types; (1) board games and (2) non-board games (such as arcade games and adventure type games). Game programs require a great deal of attention to details, especially about the rules so that, if you are converting a game, you can be assured of the completeness of the implementation. You might also want to think about the various ways in which information about the game can be communicated, both text displays and graphics. All of these considerations lead to the overall design of the program.

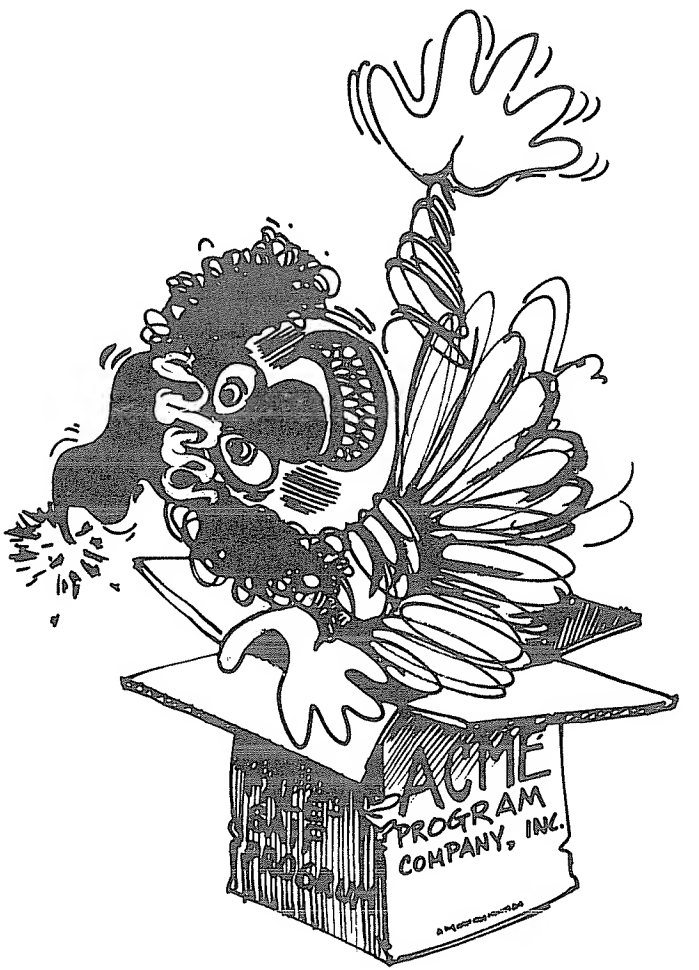
Our second category is utility programs, programs that perform functions which would normally be considered part of the system itself rather than a program you would want to run over and over again. Examples of this group would include file-to-file copy programs, disk reconstruction programs, etc. Concern here is for speed and accuracy of the functions to be performed. Communication with the computer user or operator is also important, but the detailed requirements of a game program may be avoided.

Another area that we may be interested in is that of educational software. We may develop programs to assist us in a class we are taking, or perhaps to assist our children in their studies. Educational programs may even fall into the games category, but for the purposes of this discussion we will consider them separately.

Probably the largest category for most of us will be in the area of application programs. These are programs that are designed to perform some specific task which is ususally, but not always, business related. We often think of general ledger, payroll, and other accounting programs in this category. For us, however, we can also think in terms of our own personal needs; most of the programs we will be developing for use on our own computers will probably fall into this "applications" category.

Just why this is so probably needs to be looked at in some detail. If we consider the needs for programs that we have, they will more than likely fall into the same sort of groupings as those mentioned in the various hobbyist magazines. Programs for keeping track of things; collections, records, magazine articles, etc. We may develop programs to keep track of the family budget (remember, we used that argument to justify getting the darn thing in the first place!), or to convert a recipe for four to allow us to feed a visiting army (some company used an advertisement like that in the early stages of marketing their computer!).

Although we normally place application programs in the business world, we also see their needs in our own lives. An application program is, in its simplest definition, any program that is not part of the operating system, and so therefore it is really a collection of all of the categories listed above except for utility programs. But we need to go beyond this rather broad view of application programs to see how we can get our ideas of programs channeled into logical paths.

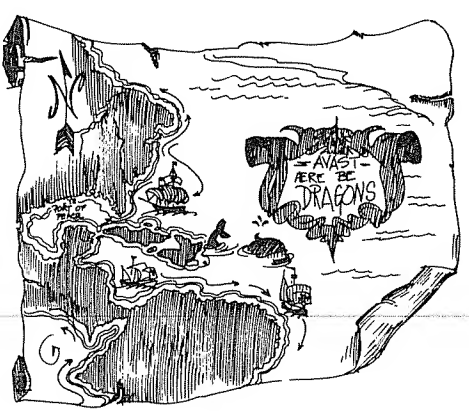


Before plunging into this, I'd like to point out that we will be discussing a technique for coming up with programs, not a fail-safe method for building programs. This set of techniques may not work for all of you, and you shouldn't feel that you are forced to stop doing things the way that you've been doing them so far. In fact, you may be pleasantly surprised to discover that you are already "doing it right!"

The techniques presented here are, to be honest, some that at first I found to be downright impossible to follow through on because they seemed so strange! But, after careful evaluation, I decided that I really didn't like them for two very personal reasons; (1) they seemed to limit my God-given right to make artistic statements (whatever that means), and (2) they forced me to approach the programs in a formal, non-personal fashion. All of this seemed to limit my own personal desires. How wrong I was!

The approach defined by these techniques actually made the job of writing programs much, much easier. It did this in several ways; I was forced to think about the program design in terms of functions and the relationships between those functions, in terms of the data flow through the program, and in terms of the way in which the computer could best perform the required tasks. The end result, not surprisingly, was a better structured program that was easier to understand and document as well as one that required less debugging (removal of program errors).

Now, having built some rather big "pigeon-holes" into which these program ideas can be placed, we can proceed to explain the reasons for doing this. Let's consider a slightly exaggerated example to make it easier to understand.



Ideas, like water, tend to flow in the easiest direction. But, and here's the real problem, that flow is often uncontrolled, or rather undirected. If we have an idea for a program, say to automate our magazine subscription processes — renewals, what we are subscribing to, costs, and so on — we have started with an idea which is applications related. But, if we take the easiest path we will immediately start writing the program.

Is there anything wrong with that? Not really, but it does show a lack of thought if we start programming (sometimes called "coding" because we are said to write "lines of code," especially when writing machine language code) without having spent some time on thinking about the "pigeon-hole" into which the new program belongs. You see, we may already have some of the routines that we need for the development of this program. They may be in a magazine

article that we've read, or perhaps they're in a program we wrote some time ago. If we haven't trained ourselves to think in terms of the basic building-blocks of programs and the common relationships between programs of similar types, we may spend lots of time reinventing the wheel.

Actually, I think that we would all remember solving a major problem that we had encountered before, but perhaps we'd not thought about it until it was too late. Although, I must admit that I have often forgotten about some simple ways to code a particular routine, and

had to reinvent it. That hasn't happened to me since I started using the methods that we will be describing in this book.

Ideas are the basis for a program, but they need to be channeled, just as flowing water needs to be channeled to be made useful. This chapter presents some ways in which ideas may be channeled, some tools to help simplify the way in which we think about programs we are about to write. We won't discuss how we transfer ideas into lines of program code, that will have to wait for Section 2, but we will look at ways to clarify our own thinking.

Let's take a simple idea and see what happens when we apply some simple "idea directing" to expand on the idea so that it can be developed into a fully functioning program. We can use the magazine tracking idea for this purpose. We'll see that this approach, although it seems to be the long way around, actually cuts through much of the leg-work required to write a program.

Our basic idea of a magazine subscription tracking program is, as we've already said, an applications program. That is, it is designed to solve a specific, real life problem that is not related to the computer's operating system, the computer itself, or any other "non-problem" related item. What does that do for us? First, we can immediately eliminate some routines from other programs, specialized disk I/O routines that are operating system independent are out, as are game-type graphics.

We might include some random access techniques, but nothing that the operating system itself can't handle. We might consider the use of graphics to accentuate renewal information or perhaps price changes (although we know that no publisher would do that to us, would they?). We will want some kind of reporting mechanism as well as human interfaces (we "gots to talk to the user," right?).

See, by simply looking at the kinds of routines needed by the programs category or "pigeon-hole" we can quickly come up with a list of routines that might be needed without doing any detailed evaluation of the program's needs. It is this kind of thinking that allows the neat and logical development of programs regardless of the category chosen.

As we shall see, this is going to be covered in some detail as we proceed, but we needed to make sure you understood the reasons for doing this. Boring? Perhaps, but so is a class in blueprint reading, and just as important! We might remember, especially at 3

a.m. while trying to get rid of the remaining bug (remember the last time, we were almost there and . . .), that the most boring job in the world is "shooting a bug" we thought we had fixed. By the way, for you history buffs, the term "bug" which refers to a problem in a

program or hardware, was coined in the early IBM days when a moth was found in a relay (big clunky things, used as switches for on-off conditions, sort of like huge bits) preventing it from closing. Getting rid of it was called "debugging" and the term has stuck. See, there is a reason for some of the computer terminology!

CHAPTER 2

Where to get Ideas

Ideas float around us everyday. There is hardly a day that goes by in which we don't get a new idea. In this chapter we will examine some of the ways to get ideas, as well as some of the places that we can look for ideas.

The problem of a programmer is the same as that of a writer in many ways. Both are creative tasks and both are highly dependent upon getting an idea. For writers, there is often a time of drought, a dry spell where there are no ideas. Usually, this is solved by sitting down at a word processor (who uses a typewriter these days?) and starting to write, writing anything at all. This is usually trash and is thrown away. But it gets the writer going. It seems that ideas always like being around activity and other ideas.

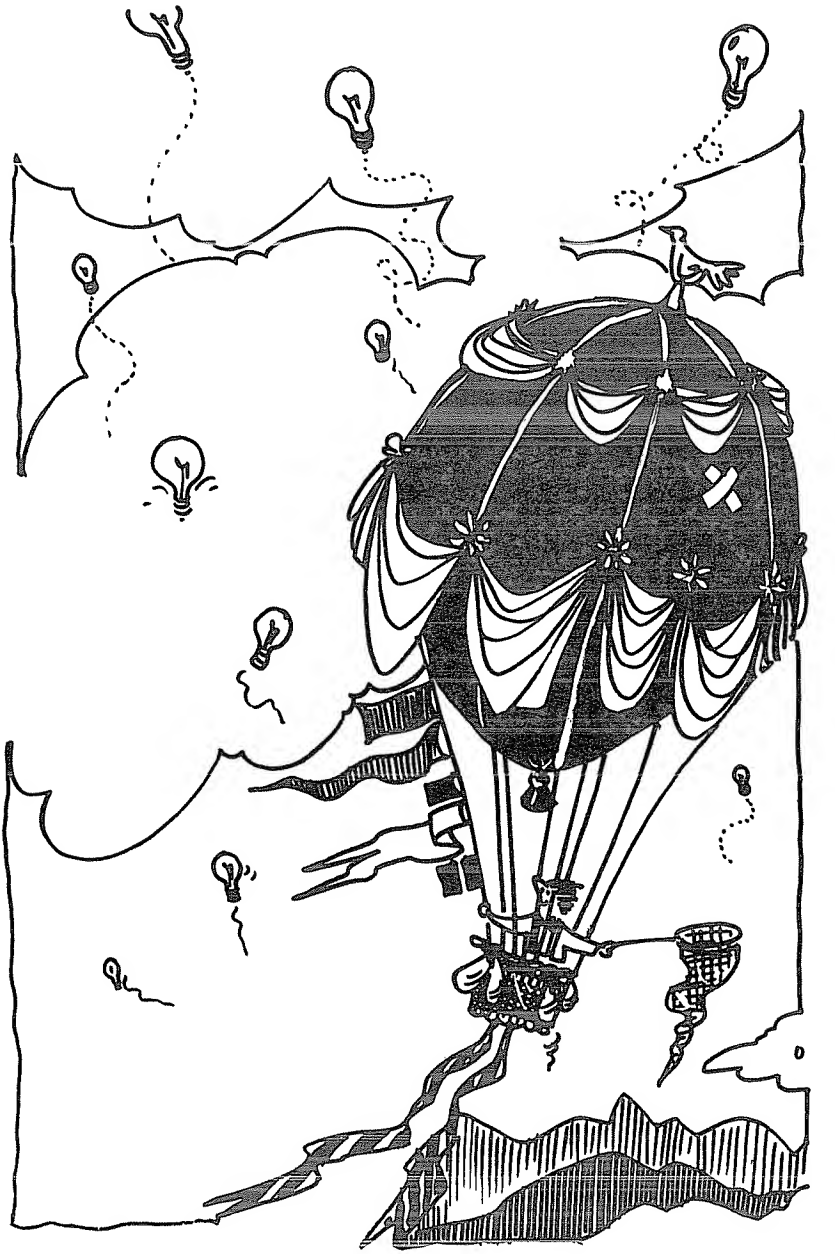
For programmers the same is true. We aren't usually lucky enough to be able to sit down and write best-selling software every day, but we can churn out stuff that we like, and the heck with everybody else! Right? Right!

Now, realizing that we are going to churn out acceptable software is not the same as churning out quality software. We are going to learn to produce only top-grade stuff, high yield programs. Even if we never sell the programs we will feel better about the program if we know that it is good, clean software.

In this industry, there's often a tendency to look at a program and say, "This one's just a quick and dirty, no need to make it clean, elegant, or fancy. Just make it work!" But if we can crank out quality stuff in the same amount of time or less, why settle for a Q & D (Quick & Dirty)? The point of this book is that it takes no more time to produce quality software than it does to produce junk if, and that's a big if, the problem is approached correctly.

How do we approach the problem correctly? Well, to quote a well worn cliche, "Begin at the Beginning." The idea itself is the begining. Picture this; we're at a movie, and the screen flashes bright with the words "THE END" just as we get to our seats. What's the first instinct? Hey — are we late? Check the watch and find out.

Why did we do that? We're conditioned or trained to expect those words to relate to the end of a movie rather than the beginning.

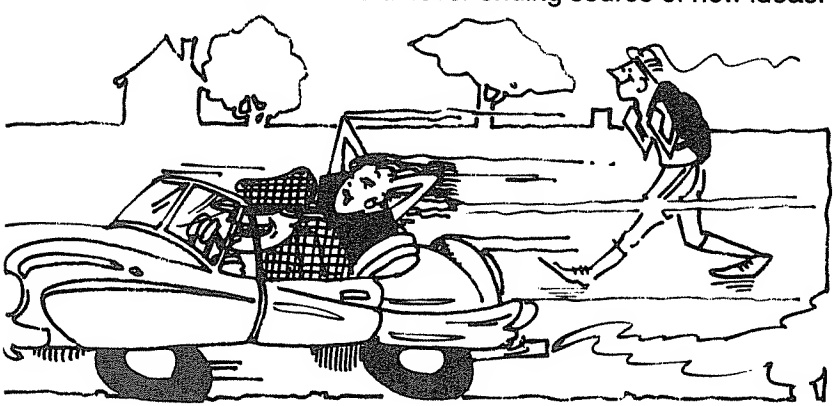


A few movies have been produced that do begin with those words, and end with "THE BEGINNING..." or some such, and they produced the anticipated results, confusion.

Programs, on the other hand, are not supposed to result in confusion, but rather in clarity. They solve a problem, and should do so in a direct fashion. If we have understood the requirements or the need for the program we should be able to code a program that will meet them. But, how do we develop these requirements? Where do they come from?

Unless you've been asleep, you already know the answer. They come from evaluating the idea; from a logical development of the concepts behind the program.

"Ok, you've convinced me. But where do I get ideas?" Very good, an intellegent question. Ideas, says the man, are where you find them. Horace Greely is reported to have said "Go west, young man." and it is indeed certain that ideas are in the west (also the north, south and east). But more importantly, ideas are in US! If we will take the time to research them, develop them and expand upon them, we will find that we have a never ending source of new ideas.



This source of ideas depends upon our perception of the world around us, how we see what we are "passing through." By looking around us with the eye of a programmer rather than just a visitor we will soon begin to see ideas for programs in the events that take place around us.

Programs are designed to solve a particular need or problem, or perhaps to stimulate us with a game or an educational program. We will be better able to develop games if we see what is going on.

Look at the first game programs — football, craps, blackjack, etc. All had one thing in common — the game already existed and the computer was being used to replace a human being so that the game could be played anytime it was wanted. Chess is an exception to this rather general statement in that it was developed more as an exercise in artificial intellegence than as a game in and of itself. In fact, that brings out another point — don't limit your views. Chess program development was rather stagnant while it was being investigated from a "scientific" viewpoint. It wasn't until people started working on chess programs "for the fun of it" that development really took an upswing. We must realize that programs like MicroChess and SARGON were quantum leaps ahead of the majority of the programs that had been developed on the huge mainframes. The people that wrote these programs couldn't depend on those blindingly fast monster computers to churn through hundreds of thousands of positions a second, so they developed a better approach, one that was suited to the microcomputer. It was, as the ad says, an idea whose time had come.

Open your eyes to the things around you, there may be many, many program ideas there just waiting to be developed. If you think of the number of programs that are under development today, you'll find one common point, all of them began with an idea based upon an observed (or hoped for) need. In some cases there has been a duplication of efforts, either because of ignorance or because of the NIH syndrom. NIH stands for "Not Invented Here" and is often a very destructive tendency. It assumes that, unless a program was written here (wherever that is), it's no good! Sometimes it's better to rewrite a program, but often it's much better to spend your time and energies on the development of a different program.

There are, of course, exceptions! If you feel that the program is a real loser, and, that you could develop a winner based upon the old program then by all means go ahead and duplicate the effort. If nothing else, it will serve as a training ground; you may learn a new trick or two, and you may actually succeed! However, beware of the old cliche "Nothing ventured, nothing gained" and make sure that you are aware of the time exposure and risks involved in modifying or rewriting a program. Will the end result justify the expense of time? It really doesn't matter whether you are doing this as a hobby or a profession since time is limited and there's no sense in wasting it in unprofitable (not necessary in the monetary sense) pursuits.

As we develop and grow in our love of computers, we often start subscribing to different magazines, professional as well as hobbyist. These are a treasure trove of ideas. Various articles in them may kick off an idea or two that you can follow up on. It's possible that a program may have a routine in it that you need to get past a stumbling block in an idea that you had for a program.

Ideas like to be around activity, and so you may be able to stimulate the flow of ideas by joining a computer club. There are many different kinds of clubs and they come in all sizes and shapes. Some examples are the Boston Computer Society and the Orange County TRS-80 User's Group (OCTUG). The first is open to any kind of computer (and is also quite large) while the second is open to anyone but concentrates on the various Radio Shack computers (and is also quite large). The discussions that flow around these clubs are quite often the source of inspiration for a new program or two. If you don't live near an area where there is a club, check the listing in some of the general hobbyist magazines, not only will you find clubs for your particular computer, but also some general clubs. Almost all of these clubs have a newsletter that is mailed out on a regular basis and it too can be the source of ideas.

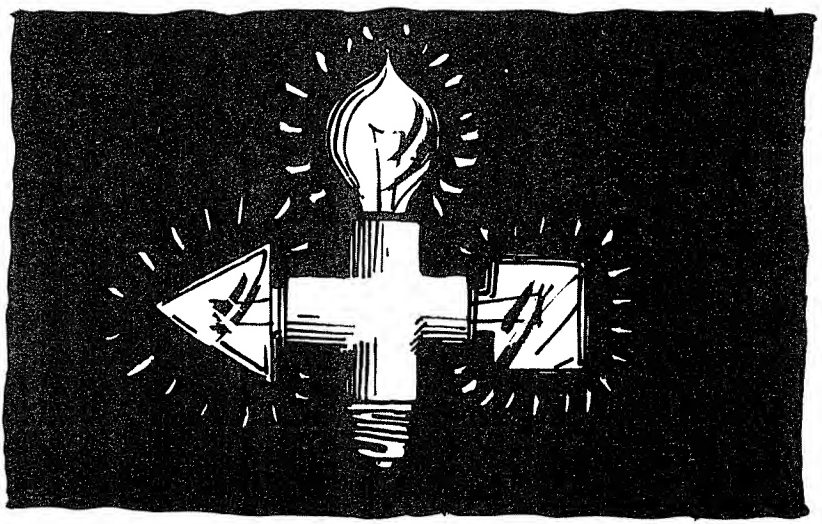
Another good source of ideas is your place of work. No matter where you work, even in the most boring job imaginable, there's always something that can be the stimulus for an idea if we will learn to really LOOK. You see, it's often the familiarity with the places around us that kill ideas! We will fail to see the possibilities because we are closed to them, too familiar with what we have and, sadly,

blind to what we don't. There's an old saying that familiarity breeds contempt, but that's not exactly true, it might be more accurate to say that familiarity breeds indifference! We become so wrapped up in the day-to-day things that we really don't see what's around us.

How often have we traveled and marvelled at the sights around us, without seeing similar things at home? That's the same symptom, just in a different setting. The bottom line, then, is that there is no place like home, and that an idea, like gold, is where you find it.

So, although this chapter is called "Where to Get Ideas," it's really all about life, and the fact that ideas are everywhere. But, and this is a big but, ideas need to be recognized. That may seem like an obvious statement, but it really isn't. More than once I have overlooked an idea for a really good program, and someone else has come up with it, and sometimes made a fair profit!

How does one go about recognizing an idea? That's a hard question to answer, and in fact, there really isn't an answer! But there are some tips that we can pass on. They aren't sure-fire by any means, but with a little work they will lead into the proper habits for recognizing an idea — whether or not it's a good idea is up to you to decide.



The first tip is a purely mechanical one. As you look at various jobs or tasks, ask yourself if they could be performed on a computer. If the answer is yes, you have an idea. There are some obvious questions that need to be asked after that point such as does it require any special hardware or software requirements. Also, what does it require in the way of interfaces to the "real world." If any of these considerations make the idea unworkable, don't just forget it, write it down. It may be something that becomes workable later as newer hardware or software development tools are released.

The second tip is actually one that is contained in the first. Write down everything that might make a program; good, bad or indifferent. Don't judge an idea immediately. Let it sit for a day or so and reevaluate it in a different light. We sometimes have a tendency to think an idea is great just because we thought of it (and that may be true) or, conversely, that it's bad because it seems bad at that precise instant.

Time, it's said, heals all wounds. It also has another nice property. It provides us with distance from our source of an idea. That distance can help us evaluate an idea with a remarkable amount of clarity. Taking time before evaluating an idea should become a rule that is never broken, never bent. The primary reason is that nature is a contrary beast, and an idea that seems good often isn't and conversely an idea that seems poor often isn't. Remember, Murphy had the right idea — if, under any given set of circumstances, something can go wrong — it will! With ideas, that means we'll spend time developing bad ideas and we'll throw away good ideas. It can surely lead to a sense of frustration.

Finally, the last tip, and the one that's hardest to follow-up on, is to never overlook a possible source of ideas. Ideas are useless if you pass them up, if you don't even look their way! We've covered a lot of area in terms of where to look. The main purpose is to help develop your own sense of where to get ideas. It's different for each of us, and you alone can determine where best to get ideas, at what level of development you will be able to recognize an idea.

It's possible that ideas will need to be reasonably well developed, and so books and magazines could be good sources. As your sophistication in picking up ideas grows, you should be able to spend more and more time getting ideas from less well developed sources, say from discussions at club meetings or with other computer hobbyists.

One final point, don't limit your reading material in the computer field. It's said that the best way to learn English is to read, read, read. The same is true of computers. You can best expand your knowledge in that area by reading (as you are doing now). The more books you read on the subject, the more facile you will become in getting, recognizing, and developing ideas. And that, after all, is the name of the game.


RUSH

CHAPTER 3

Combining Ideas

In this chapter we shall take a look at how some relatively simple ideas can be combined to form a complete picture of a program. We will be looking at various ideas which may be combined by you to form programs as well as the ideas which will form the basis for a program that will be developed throughout the rest of this book. So, pay attention and take notes, there might be a quiz at the end of this chapter (Yuck!).

We'll begin by taking some VERY obvious ideas from the world around us. Let's take, for example, scheduling our time. As students we might have to worry about classes, homework, dates, extracurricular activities, dates, football games, dances, dates, etc. It might be nice to make sure we don't schedule a date with two different girls (or guys) on the same night. Right? Or perhaps you're a business man and need a way to better manage your time, again a scheduling program might come in handy.

In any event, most people in business and the military have been faced with the idea of time management, or how to avoid wasting their valuable time. There are many programs that are quite involved and complicated being released now to serve this purpose and all were based upon the idea of a computer based calendar/scheduling program. So that you can develop your own version of such a program, let's take a look at what might be involved.

There are some obvious requirements. The program must be able to keep track of the day of the week and the date. It should be able to make entries for various times of the day and should warn you of scheduling conflicts (places where two or more events are scheduled at the same time). It might be nice to have entries that are "memo" type notes rather than scheduling information.

How about that? From a rather simple idea, one that has kept a lot of companies in business marketing calendars with lines for the various working hours of a day, we have derived some simple requirements for a program. If you've looked at any of these "work-a-day" desk type calendars, you'll notice that the weekdays have two pages — one divided by hour and the other blank for making notes. Just exactly what we were discussing! There was no reason to be esoteric or to really strain our brains trying to come up with exceptionally difficult design criteria; all we needed to do was to look around us and keep our eyes open.

Now, let's go one step further. What happens if our schedule changes (we're all perfect, but some of those we deal with aren't and THEY cause the changes)? Well then, we need a way to change and delete entries from the calendar (my, my — sounds like an eraser). Still, nothing out of the ordinary, we're just looking at how we really solve a problem and trying to define how we would do it on the computer.

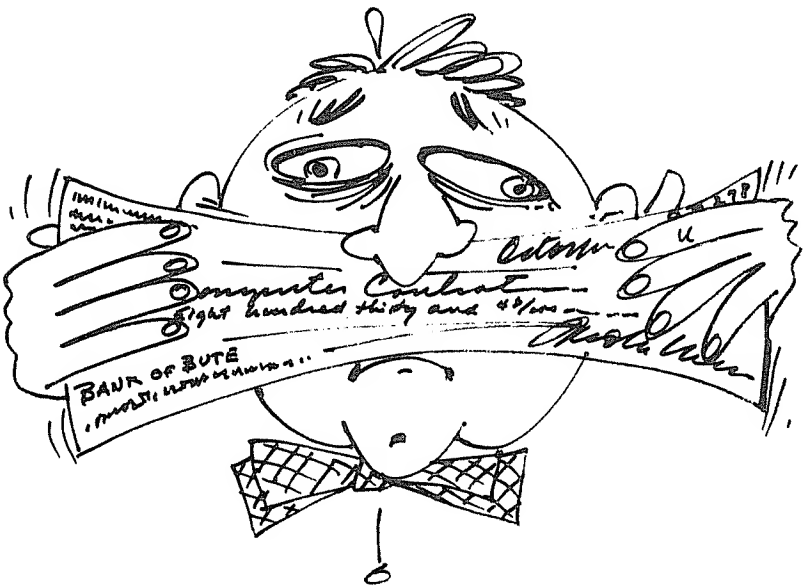
To make sure you have gotten the point of this chapter, let's review what we have said, and see how it fits this example. We've been talking about the combination of ideas, taking several small ideas and combining them to form a more complete program (we'll talk more about that later on in this chapter). The ideas we have used are:

1) A Calendar.
2) Entries by day/time.
3) Ability to flag scheduling conflicts.
4) Update/Delete capability.

Have we omitted any ideas? Actually, what's been omitted is not so much an idea as a program requirement — reporting! We need some way to get the information out, perhaps in hard-copy format. See how easy it is?

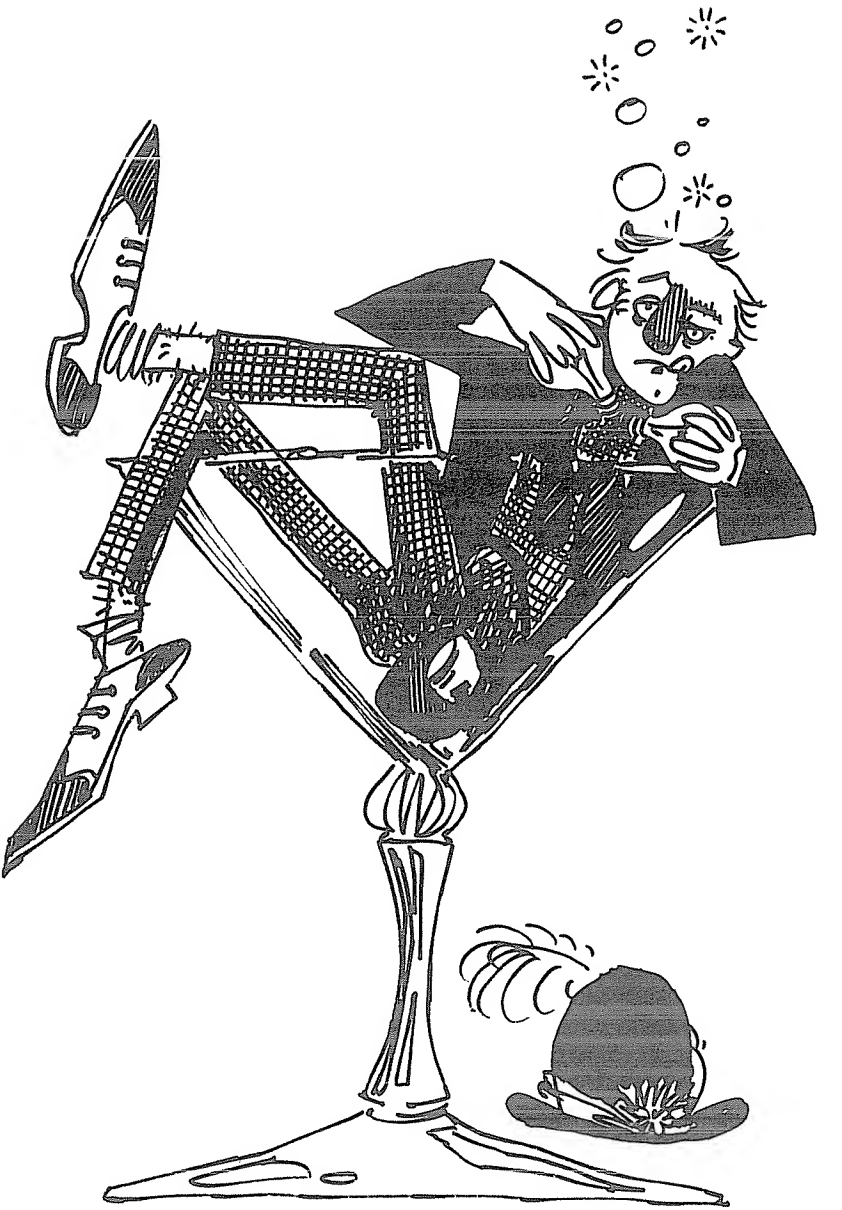
Let's continue this discussion by looking at the idea of a checkbook. What do we do with a checkbook (besides writing rubber

checks)? Right. Sometimes we write real (that is, good) checks. What about service charges? Do we get interest on our checking account? When we write a check, do we ask ourselves if there is sufficient money in the account to cover the check? Do we write the check anyway?



In this simple paragraph we have discussed a number of small ideas, none of which are large enough to make a good program all by itself. But, by combining them, we have a basis for a large, relatively sophisticated program! We'll come back to these ideas later — they'll form the basis for the program that we will be developing throughout the remainder of the book. But for now, let's continue to look at ways to combine ideas, some guidelines for doing various forms of combinations.

First we'll take a look at what we mean when we talk about "combining ideas." We've seen some examples of combining ideas, one in some detail when we talked about the calendar program, but we need to dwell on this some more. Combining ideas is a lot like driving and drinking — dangerous! In the case of ideas, it's not fatal to you, but it may be to your program! Ideas can usually be combined when they have some relationship, some common ground between them. Sometimes ideas that are not directly related may be combined to form a program. For the purpose of building some pigeon-



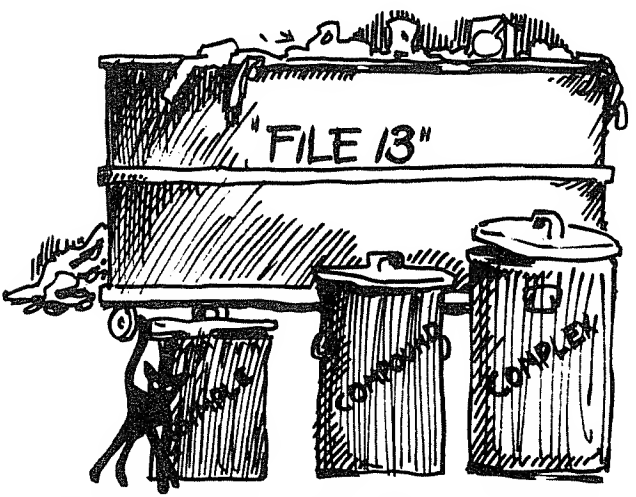
holes to hold our "combinations," let's try the following:

Simple Programs — Based upon one idea only.

Compound Programs — Based upon two or more related ideas.

Complex Programs — Based upon two or more unrelated ideas.

This will provide us with a way of talking about programs that is not overly confusing. But (why is there always a but?), for those of you who might have written programs before reading this book (there's always at least one in every crowd), we might want to remember that there's no such thing as a simple program, some are just less complex than others. So, when we are using the terms "simple," "compound," and "complex," let's keep in mind that it's for the purpose of this discussion only, and does not relate to the "real world." Ok?



Now, let's examine some obvious combinations of ideas. The first that comes to mind is a set that is almost always considered to be one idea rather than three. When we write a program that uses any kind of data entry, we usually provide entry, update, and delete routines. The tip-off that this is really three ideas should come from the fact that this is usually implemented as three separate routines. This provides us with a rule of thumb (a guideline that is stated as being obvious), "One idea per routine." Therefore, our defined "simple" program is really only a single routine.

Nothing's ever THAT simple! There are usually support routines necessary for every program, and they are not usually considered

as separate ideas (but they are). Yes, a question from the back. Oh, you want an example? Glad you asked! Screen input routines may be considered as merely ways of getting data into the program; but they can actually be very complex, especially if we are using "full-screen" type data entry routines. In this case, the term "full-screen" is used to indicate that we have a "fill-in-the-blanks" format on the screen and, under program control, the user is entering data which is being verified and echoed in a way that simplifies data entry.

Obviously, using routines like that require some thought, and are usually written as subroutines. Remember, one idea, one routine. What this does for us is to allow the development of a common "full-screen" routine that can be modified and dropped into the program, thus we don't really have to worry about the particulars of the routine, just the appropriate changes. This will make the program development go faster since; (1) the routine is available, and (2) it's already been tested so we only need to verify the changes.

Other examples are report printing routines, file handling routines, sorts (there's an example of a good sort in the appendix), ETC. Once these routines have been developed they should go into a library for future use. There is no sense in doing something over and over when that's what computers are good at (and they're faster at it than you are). Let the computer help you develop good programs. After all, you're the boss, not the employee.

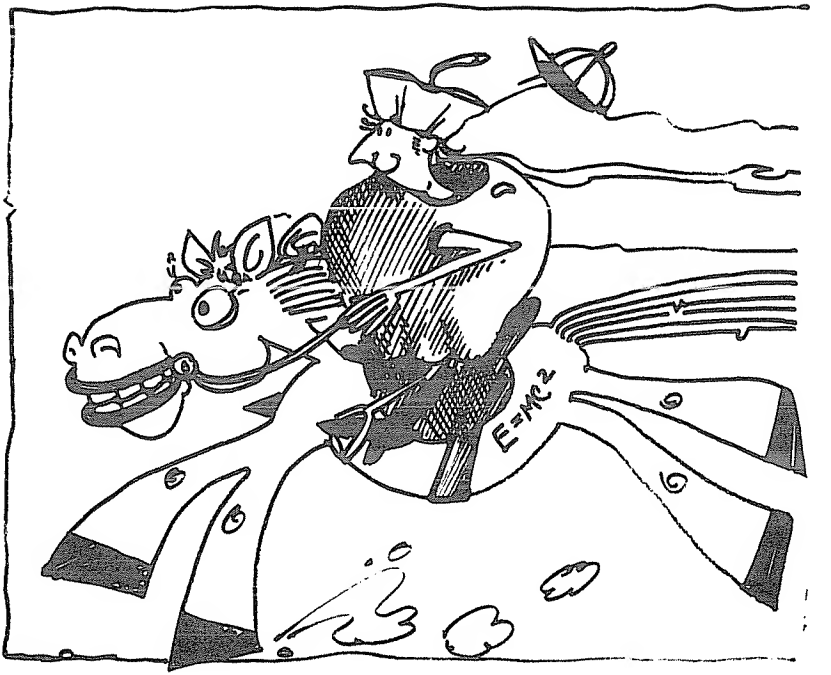
Ok, what about compound programs? Well, to be honest with you, almost every program is a compound program in that it has several related ideas present in the form of support routines. But that's not really the point that we're trying to make. Instead, let's consider our calendar program again. It should be easy to see the relationship between the various ideas. This is actually a compound program because there are many ideas, each of which would form a single routine.

By the way, let's pause for a moment and discuss the difference between a routine and a subroutine. A routine is the physical implementation of an idea, while a subroutine is a block of code used in many different areas of a program, and coded once for simplicity, compactness of code, and ease of testing/debugging. Easy enough? Don't worry about that now as we will take another look at these two terms later on in the book. For now I just wanted to make sure we were all talking about the same thing.

Now, what about a complex program. Are there any simple examples of a complex program? That's not an easy question to answer, but we'll take a look at what a complex program is, and see if maybe there might not be some examples from everyday life that we can draw upon. By our earlier definition a complex program is one that is based upon two or more unrelated ideas. The key to this definition is the letters "un" meaning not (after all, that's the only difference between a compound and a complex program).

We know what an "un-cola" is because of our exposure to the television advertising people, what's an "un-relative?" Let's take a simple example which is, I'll admit, a bit absurd. Suppose we wrote a program that would determine the winner of the sixth at Santa Anita and, as a separate function, determine the maximum possible energy that could be released from an apple weighing a half kilogram (roughly 1.1 pounds) utilizing the standard expression relating energy to mass ($E = mc^2$) with the results expressed in ergs.

Alright already, so we can come up with bizzare examples



(actually, it's mandatory to do so, failure to pose ridiculous examples can result in being kicked out of the Society for the Advancement of Dumb Examples (SADE)). Now, let's try something a little less far out! Suppose you wanted to write a Real Estate package that would provide a local office with a listing of all homes in the immediate area. This listing would include prices, size in square feet, number of bedrooms, relative location and so on. In addition, the program would keep track of the various lending institutions' current rates and policies.

This, although it seems to be related, isn't. They are both lists, and so in that sense they will fit together, but the type of lists and the source of the data is quite unrelated. The resulting program might be quite useful, but would indeed fall under our definition of a complex program. This definition, as was pointed out earlier, is not one that fits well in the real world and is really only useful in discussions of this sort where we can put blinders on and ignore certain aspects of the way it really is.

I certainly hope that you will keep this in mind as we continue through the book. Places where we refer to these rather simplistic

definitions will be pointed out so that it doesn't confuse you (or me). We will continue with our discussion of program development, especially as it concerns the development of our example program.

One last point before we go on to Section 2. The material that was presented here in this first section was designed to get you thinking in a fashion that might have been foreign to you. The reason for this is quite simple. For most of the past 25 years of computer programming activities, we've been doing it wrong. It's only been within the last several years that an attempt has been made to systematize (here's your new word for the day, it's another fancy word that means making an orderly classification) the development of software. Several new branches of software development have been formed, the most promising being called "Software Engineering." This term is used to refer to the underlying techniques used to develop software and the attempt to bring the more concrete tools of engineering to bear on the problem.

The theory is that this will allow development of better computer programs that take less time to write and debug. While this may be just a nicety for the hobbyist, it can also be a boon since the average hobbyist doesn't spend all of his time playing with his computer (I understand there are even some hobbyists that actually sleep!).

So, we've spent a moderate amount of time developing the understanding of the background of programs, the idea. And we've taken a look at some of the different sources of ideas in the world around us. We've gone to some lengths to develop a couple of ideas into rather primative program design statements which could lead to the development of a couple of quite useful programs.

But, and this is more important that all the rest, we've provided a framework for your thinking. We've learned that we can channel our thoughts so as to maximize the results and the time spent, not in a 1984 sense of "rightthink" but in a more disciplined sense of personal development. I like to think that we are trained to think so that we can improve on our training. That is exactly what I have tried to provide in these first three chapters. And now, let's see if it will really work!

Section 2 — Planning the Program

In this Section we will be addressing the way in which we put our ideas together. There is a right way and then there are some other ways, not necessarily wrong. Don't misunderstand me, we aren't going to be grinding any axes here, just looking for the most efficient way of getting a program written, debugged, and documented in the least amount of time, with the least amount of difficulty. Sounds simple, doesn't it? Well, it isn't!

There are lots of ways to put programs together and each programmer will develop a technique that is unique. What we are going to try to do is show how you can take your own technique and, with the tools given here, expand them into a better, more efficient technique. One that is still yours, but one that has been tried and tested by the professionals.

CHAPTER 4

Program Functions

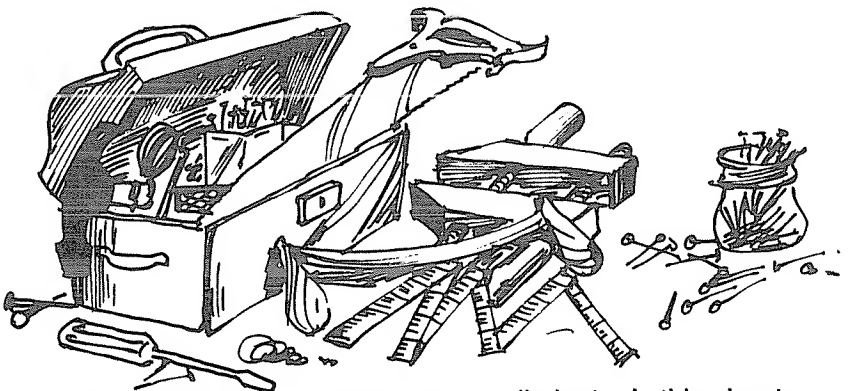
As we begin this chapter, let's make sure we understand where we are going. An old adage for teachers is that you must tell the students what you are going to teach them, teach them what you have told them you are going to teach, and then tell them what you taught them. We'll use the same approach for this book. Here I'll tell you what I'm going to teach you, then we'll actually learn the material. Then we'll go over what we learned, that is, I'll tell you what I taught you. Make sense? Seem fair? Good!

As with everything that we will try to do throughout our lifetime, programming is something that needs to be learned. There are very few things that we do that are instinctive in the way of human activities. With the exception of the life preserving functions of breathing, pumping blood, maintaining the proper biological necessities of life and the basic flight/fight reactions, most of our activities are acquired through some aspect of learning.

What we are about to undergo is a form of learning, a more typical example to be sure, but one that tends to be offensive to some individuals. We will be learning to place a scientific framework


THERE
IS NO
TOMORROW!
CEM
THINNER

around an activity that has been considered an artistic endevor. Writing programs is, at best, a challenge and at worst an encounter with frustration and hopelessness. How often have we started writing a program that seemed so easy when we started and turned out to be one gigantic pain in the neck (or elsewhere)?



It's that pain that we are going to try to eliminate. In this chapter we will be looking at how to go about defining the functions that are needed in a program based upon the ideas that we have. It is only by doing this religiously, without fail, that we will be able to minimize the frustration felt by starting programs that wind up becoming either too big to handle, to unwieldy to maintain, or both! Without further ado, let's press on, shall we!

If you will recall, we started this book by promising to develop a program. In Chapter 3 we discussed some of the ideas we wanted to implement in the program we were about to develop. These ideas related to a checkbook program. Now, at first glance, there doesn't seem to be anything very exciting about developing a checkbook program. After all, that can be done quickly and simply with a pocket calculator, right? Wrong!

We're not going to waste our time with a program that is too trivial to even think about. Let's go over some of the ideas that we had wanted to implement and, as we go, we'll take a look at the function (or functions) contained in the idea. Ready? Actually, you'd better be! It's too late to turn back now!

The first idea we had when we thought about checkbooks was that we write checks, both real ones and (for some of us, occasionally) rubber ones. Ok, function one is to write a check. For the time

being we'll assume we have a printer that can print on regular checks so that we don't need to order continuous form checks. See, very simple. One idea equals one routine to print checks.

"There's probably more to this programming stuff than that," says you.

"Right!" says I.

That's just the title of a functional routine in the program since there's no BASIC statement called PRINT CHECK (at least there wasn't the last time I looked). Actually, I'm lying! That's almost a perfectly good statement, and it's possible that the variable CHECK may be defined in such a way as to contain the data we want to print, already formatted and everything! See what I mean? Nothing's ever as simple as it seems!

"Wait a minute. If we're going to write a check, aren't we going to need to see if it's going to be a good check or a rubber one?"

Good point! I'd almost forgotten that. Another routine or two will be needed to check the amount and tell us if we can cover the check we're writting while another will decrement the balance (that's a fancy way of saying subtract the check amount from what we've got left in the bank). And while we're at it, we might want to log who the check was written to, the amount, and the date. Boy, this is sure gettin' complicated!

Don't despair, you'll see how easy this is when we actually write this program.

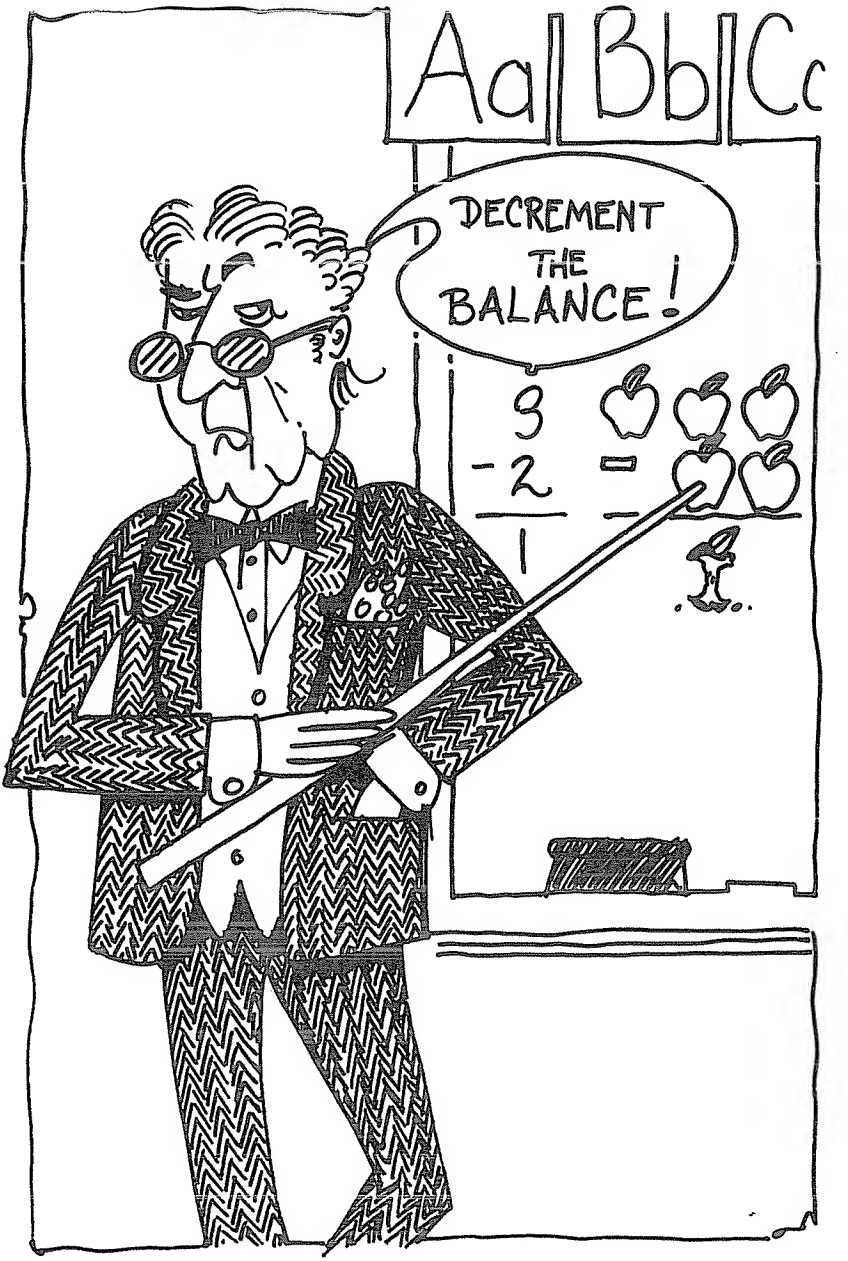
"What? We're gonna' write this turkey?" you ask, incredulously.

"Yup! And you're gonna' love every minute of it!" says I.

"Not hardly." you mutter under your breath.

To continue developing our program, let's assume that we are all rich enough to get interest on our checking accounts. We know that the interest rates are public knowledge and we are also able to get information from the bank regarding the way in which the interest is calculated, the posting dates, etc. With this information we can write a routine to implement the interest calculations. Again, we are taking the idea of interest and making a single routine from it. I don't want to dwell on this too much, but it's importance can not be understated — never, never, never, make a routine that contains more than one idea (or function). The results can be terrible! Debugging, maintenance, and testing can be made far more difficult than they need to be if you ignore this very simple rule of thumb.

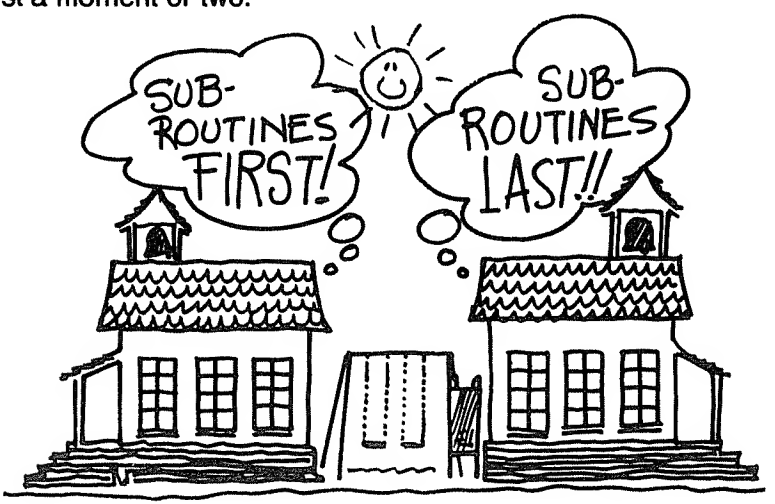
What, a complaint from the back? Yes, I know that this will


Aa Bb Cc
DECREMENT THE BALANCE!
3
-2
1

reduce the artistic freedom to create a program any way you want. But, stop and think about it for a minute. Are you creating a program for the fun of writing it, or are you trying to develop a running tool? If your answer is the former — go ahead and do whatever you want to do. It's your time and up to you to do whatever "turns you on." But, if you are doing this because you need (or want) the program, then try following the guidelines we're covering here. They'll help! Really they will!

Enough of the soapbox, let's get on with the business at hand. We've taken some examples of the sort of things we are talking about when we talk about program functions. What else is there to talk about? Actually, there is very little left to discuss except the possibility that we want some rules or guidelines regarding functions. Especially their organization and placement in a program.

Fair enough! Since the topic of this chapter is program functions, it seems reasonable that we discuss more than just the functions needed in this program, although we will come back to that in just a moment or two.



How should program functions (or, to use a better term, routines) be organized. There are two schools of thought on this and, since there doesn't seem to be any really strong arguments supporting either school, we'll look at both and choose the one that seems better to us.

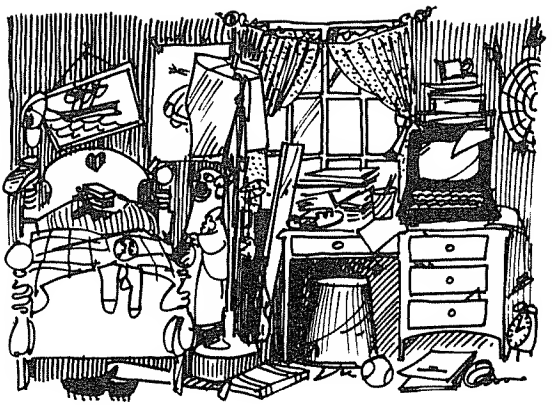
The first school of thought is oriented toward languages like PL/1 and XPL (a dialect of PL/1 developed at Stanford University).

Both of these are compiled languages (a compiler is a program that takes source language statements and converts them into machine language statements directly executable by the computer without the need of an interpreter such as BASIC). The major difference is that PL/1 is usually what's called a "multi-pass compiler" while XPL is a "single pass compiler." You'll see the significance of that in a moment.

Both of these languages suggest that all variables be defined first. This is then followed by all of the subroutines, from most minor to major, and then the mainline code or program statements. For a single pass compiler this seems to make sense since all names referenced later on in the program should be defined before they are used. The exception to this is GOTO's that jump forward in the program. There isn't much that can be done about that. With PL/1 this isn't a firm requirement since it can make multiple passes of the source file, but it's still a good idea from a maintenance standpoint. It makes it very easy to follow the flow of the program and to see how variables are defined if this all takes place in one location in the program.

The other school of thought, seen especially in COBOL, is that the variables are defined first, followed by the mainline code which is then followed by the various subroutines. Again, since most COBOL compilers are multiple pass compilers, this seems reasonable, and makes sense when the requirements of the language are considered. Especially since the structure of the language *requires* that the DATA DIVISION (where the variables are defined) occur before the PROCEDURE DIVISION (which is where the routines themselves are located).

Now, you might ask, does BASIC have a tradition? YES! There is indeed a tradition associated with BASIC. Since the language was developed at Dartmouth over 15 years ago as a language for teaching beginning students the fundamentals of programming, it has probably been subjected to more mistreatment than any language in history. The only tradition that BASIC has is that variables are defined anywhere the programmer wants to define them; and that routines are scattered all over the place. Sometimes routines are duplicated or, worse yet, split up and dropped, a line at at time, throughout the program. We certainly don't want to emulate this rather shady practice. And, if we will think back, all of us have had a program that looked like that, although it may not have started out



that way it probably did wind up looking pretty shabby, even if it did work.

Ok then, given all of these different approaches, how should we structure our programs? Well, as a rule of thumb, it makes sense to define all of our variables first, that's one thing all of the various development techniques have in common and so it *must* be good. Next, always try to structure code so that the routines are self-contained. If that's impossible, then make sure that any subroutines are called in a consistent fashion.

As far as routines are concerned, many of the experts in the field now consider it best if all entries to a routine take place at the top, and all exits take place at the bottom (physically) regardless of the type of processing that takes place withing the routine itself. This makes it very easy to change, without the possibility of destroying a working routine.

As you can see, all of these "rules" really aren't rules but are simply guidelines. You may already be doing some of these things and some you may not want to do. But, don't refuse them just because you've never used them before or because what you do now works. Those aren't good reasons for not trying something new. They're just excuses to not grow and develop. Since you're reading this book, we can assume that you are interested in being a better programmer (or learning how to be a programmer), and are therefore willing to learn and experiment. That's what it takes. None of the material that we will be discussing in this book is radically new and none of it is whimsical. All the material comes from a lot of years of work in the field and listening to what the so-called experts have to

say. Some of them are quite knowledgable and some aren't, but all mean well.

In putting together this book, I've tried to make sure that the material presented here is good, accurate, and reasonably workable in the hobbyist world. I've not concerned myself with the needs of large scale, multi-man-year projects (although the material lends itself to that very nicely). What's here shouldn't even be interpreted as things that I always do! Each program has its own unique needs and one should never approach a program assuming that there's only one way to write it.

But I digress! Back on the ranch, we see the cow . . . Whoops, wrong book.

In defining the structure of a program, especially when talking about program functions, it might be nice to provide a few examples. First, let's take a simple program that will calculate the area of a regular four-sided figure (called a rectangle, of which a square is a special case).

The program must have an input routine, a calculation routine and an output routine. The support routines would include clearing the screen and processing errors (exceptionally large numbers that might cause overflows as well as the entry of a length or width of zero which is, of course, invalid). Program termination might also be another special support routine but only if it is other than the "END" statement. Instead of BASIC, let's use a form of language called "pseudo-code" to define this program. We'll cheat and use this as an introduction to the material that will actually be presented in Chapter 7.

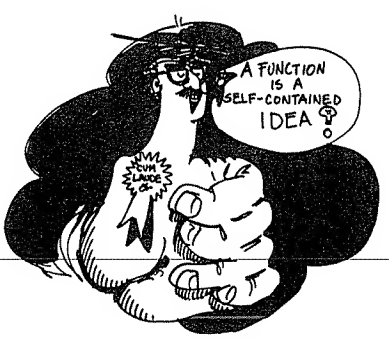
The way in which pseudo-code is used will become quite clear very quickly and, therefore, there is no need to provide more than a sample as an introduction. Therefore, the following is our program, carefully structured by functional block.

```
BEGIN PROGRAM.
     CLEAR SCREEN AND DISPLAY PROGRAM
        TITLE.

GET INPUT.
     PROMPT FOR INPUT THEN READ WIDTH
          AND LENGTH VALUES.
     IF BOTH WIDTH AND LENGTH ARE ZERO.
```

```
            USE THE TERMINATION ROUTINE.
      IF EITHER WIDTH OR LENGTH IS ZERO,
            USE THE ERROR ROUTINE.

CALCULATION ROUTINE.
      CALCULATE AREA BY MULTIPLYING
            LENGTH TIMES WIDTH.
      PRINT THE AREA AND GO PROMPT FOR
            MORE INPUT.

ERROR ROUTINE.
      DISPLAY ERROR MESSAGE.
      GO GET MORE INPUT.

TERMINATION ROUTINE.
      CLEAR SCREEN THEN DISPLAY
            TERMINATION MESSAGE.
```

As you can see, this is a very straight-forward way to define a program at a moderate level of detail but not quite at the program level. We'll use this technique more in Chapter 7 where we will fully develop the ways in which this can be used.

Before we get too far into this chapter, it might be nice to define a program function so that we will all have the same ideas in mind as we go along. A program function is not the same as a routine although it may appear that way. If we stop and think about what it is we talk about when we mention a function, we often think about the DEF FN statement in BASIC. This statement allows us to define a single statement function that may be referenced by its name alone.



That's actually a pretty good description of a function but it's not a good definition. A function is a wholy contained implementation of an idea or part of an idea. That is, it may be a routine, or it may be a single unit of code that produces what might be considered a "kernel" of information processing. That's pretty abstract, but what it all boils down to is just this; a function is a group of instructions that perform some defined unit of work, usually related.

"Ok", says you, "what'd he just say?"

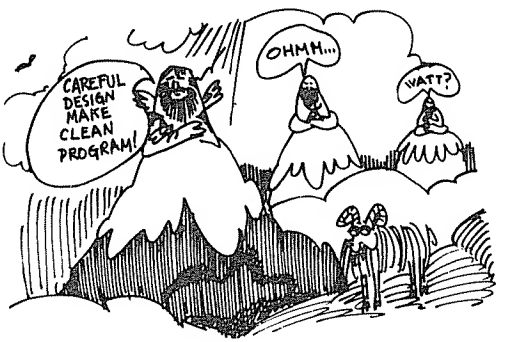
Let's take some examples and see if maybe we can clarify the issue. Suppose we are writing a program and in that program we need to calculate the remainder of a number divided by another number (called modulus arithmetic). That isn't really a routine since it is needed as a part of a routine but, it may be needed in more than one routine. We could build it as a sub-routine called by the various routines that need it, or we could build it as a function.

In this case, it's quite clear what a function is. This chapter is not just about that kind of function, but of the broader, more general definition of a function. In this case, we talk about the limited definition of function, as well as routines and subroutines, in terms of how to structrure the program.

At the beginning of the chapter we talked about where to place the various routines and data definitions. That is the primary thrust of the chapter, the logical structuring of a program. It is only through a positive, well thought out structure, that a program becomes easily maintained and understood.

If you've been writing programs for any length of time, you may have had occasion to go back to some program written a long time ago and found that you had difficulty in understanding the flow and structure of the program. That is probably due to the fact that you didn't spend a lot of time in thinking out the structure of the program.

Just as an author must think about the structure of a book, the placement of the chapters, and the flow of the information contained in the book, so too the programmer must think out the placement of the routines in his program, the flow of data through the program. This is important not just for the author of the program but also for whomever may use the program later. This is especially true if you are planning on marketing the program.

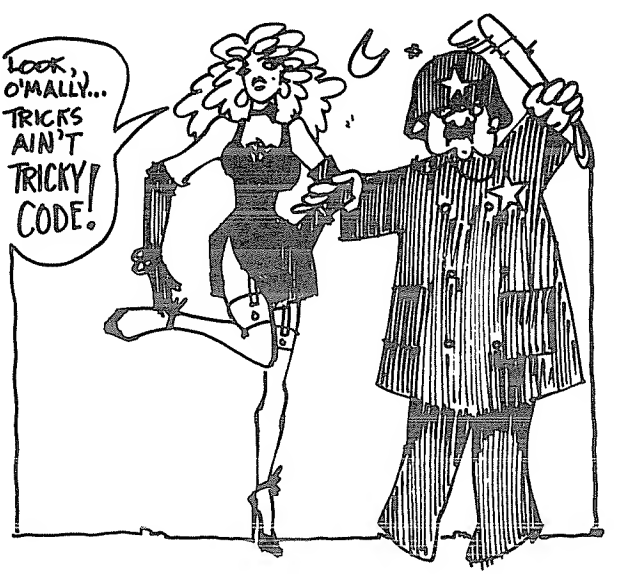


A commercially marketed program has certain requirements that far exceed those of programs written for one's own use. A commercial program must be fully documented and easy to maintain (as well as being bug-free). Additionally, the program that is being marketed has some other considerations of which speed of execution must be one. Who wants to sit and wait for a program to perform some function? Careful design of the program will allow the maximum speed possible, given the limitations of the machine and the language used.

There is, of course, another consideration. If we are marketing a program written in BASIC and we are shipping the program in source form (that is, a non-compiled program), we may want to allow the user or purchaser of the program to customize it for his own needs. This again requires that the program be logically structured and easy to follow. After all, we're all expert programmers but those that will be buying our products may not be and they'll need all the help they can get!

Sound reasonable? I certainly hope so! But this is all theory, how do we go about putting it into practice? That's not as easy to do, but there are some simple mechanical appraoches that will make the job a little easier. Keep in mind, however, there is nothing that will make up for shortcomings in the program design. No mere mechanical trick will overcome a poor program design or implementation. So, regardless of whether or not you plan on marketing your program, it makes sense to make the program the best that you possibly can. Not only for your own enjoyment and protection later on, but also because you might find out later that your program is a marketable commodity, and it would surely be difficult to go back later on and fix the program to the point where it can be marketed *and supported!*

We must never forget that marketing a program is a little like having a baby! The program (or, more accurately, the user) must be nursed along and cared for. Marketing a program requires that you be willing to take the responsibility for your product. Even if you market it through a company such as Datamost (the publisher of this book), you may still be called upon if there is a problem, and it certainly behooves you to be able to fix the problem as quickly as possible so that the publisher can get the fixes out in a timely fashion, and so that you can get on with whatever you were doing.



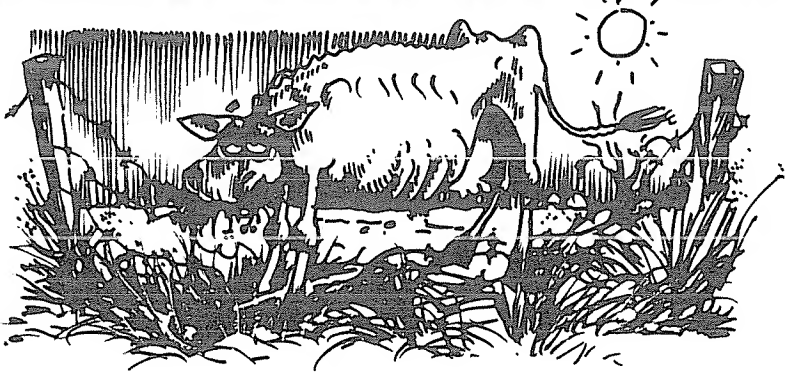
Let's talk a little about some of those mechanical tricks that we had mentioned above. And, as we do so, I ask you to keep in mind that these are just that, tricks, not tricky code. There is a difference! Tricky code is defined as code that does something other than that which is obvious. This should always be avoided, even if the code that is used takes a few more bytes or an additional few lines of code. The resulting clarity more than makes up for the extra size. Also, programmers use tricky code to show how good they are but you and I, as really good programmers, are so secure in the knowledge of our own skills that we don't need to prove it with tricky code.

Now, on to the tricks! Our first consideration has to do with the way in which the computer executes instructions. Sequentially, right! Since the normal flow of the program is from the top to the bottom, it makes sense to utilize this logic in the construction of a program. Lay out your code so that the normal path through the code runs from top to bottom in a straight-through flow. This will minimize the problem of code that jumps all over the place and is hard to follow. Then, for the unusual case or the exceptions, jump out of the normal flow. This jump may be a GOSUB to a routine that will get missing information, prompt the user for some action, or some other function where control will return to the program at the next instruction.

In the case of an error that terminates the logical flow, then a GOTO is acceptable and sould be used in a clear fashion so that when looking at the program later on you will still be able to understand the reason for the code break. A GOTO at the bottom of the normal flow is also acceptable and should be used. If you are working with languages that allow either a DO...UNTIL or a DO...WHILE construct, then that could be used with the termination condition being set when the condition which signals the end of processing is detected. This is more in keeping with the spirit of "structured programming" but, because of language limitations, is not always available to users of BASIC. Another conditional statement that might be available is the WHILE...WEND construct from Microsoft. This allows execution of a loop WHILE a given condition is true.

By the way, I've been assuming that the program's main line of code will be repeated more than once and so some kind of looping condition must exist. When writing a program, it is essential to consider the loop controlling factors. There are several reasons for doing so. The most obvious is to allow the program to correctly determine

when processing should terminate, and the other is because programs tend to fail most often at the so-called "boundry" conditions.



Just what do we mean by "boundry conditions?" The simplest definition is seen in processing data from a file where the first "READ" is the initial boundry condition and the last READ is the terminal boundry condition. In each of these cases, it is necessary to do some special data handling to correctly establish processing. For example, if you are checking the sequence of input, you need to make sure that the hold area where the initial value will be stored is initialized to a value *lower* or *higher* (depending upon the expected sequence) than any that can exist in the file. Also, if you are summarizing on some key, it's necessary to make sure that the key is initialized on the first read without triggering any of the normal non-match conditions. By the way, when we talk about *keys* we are refering to a data field within a record that can uniquely identify a specific record. Some examples may be Social Security Numbers, bank account numbers, drivers license numbers, license plates, etc. Although these are all called "numbers," it's easy to see that letters will also work (as exemplified by license plates). At the end of the input, we need to make sure that the code can treat end-of-data as a non-match condition to force the performance of the normal non-match processing as well as the end-of-data processing.

Because of the uniqueness of these two conditions, there is a greater likelyhood of a program failing in these two areas than in any other condition. All of these problem areas are going to be discussed more fully in Section 5 where we will talk about testing and debugging.

Some other considerations that should be made relate to the question of speed. Because of the way in which the BASIC interpreter works, it is best to define all variables by placing data in them *as soon as possible* in the program. On the other hand, defining their usage via the DEFINT, DEFSNG, DEBDBL and DEFSTR parameters is very useful and should still be done, especially since using the appropriate TYPE flag ($, %, ! and) uses one additional character and takes a little longer to scan. Of course, the explicit

declaration of the variable throughout the program does make its usage clear, but sufficient documentation will compensate for this single compromise with the language.

Clarity of thought is important in the development of any creative work, but especially so in the development of a program where the computer will become an extension of your brain, carrying out the tasks you have ordained for it. This can be quite an accomplishment, to have such a powerful ally in the processing of data. But, if the program is not well thought out and properly constructed, the program can be handicapped in a severe fashion. It may function poorly, it may be slow in execution, and may actually produce erroneous results.

All of these are conditions that plague those installations of large computers which insist on development of programs in the minimum time possible or who refuse to spend the time (when it's available) to go back and "clean up their act" so that they can maximize the results of their programming staff and the computer facilities themselves. I've often thought that they should be forced to develop programs on a micro so that they would realize what a limitation machine size and speed is and would therefore utilize their big, powerful machines to their fullest. Ah well, I do dream a lot, don't I?

Where does all of this leave us? By developing our thinking before we develop the program, we are in a better position to build the program. We will know what the program must do and have an idea of the various component parts the program must have to perform it's intended functions.

Suppose we didn't know what a car was and all we had in mind was a means of transportation for one or more people. We can think of all kinds of machines that will fit that bill — even today we have trains, cars, buses, airplanes, motorcycles, bicycles, tricycles, unicycles, etc. All of these are different implementations of the same idea and all are different; they look differently, they are designed with different criteria, and they are implemented differently.

Computer programs provide us with one of mankind's most interesting challenges. Even for a given trivial program, there will be as many implementations as there are programmers to implement it. This is due to the "artistic" side of programming. Since each of us will perceive a problem differently, we will each approach it from a slightly different tack, and will consequently produce a different program. It is this individuality that makes programming so much fun. The program that you write is solely the product of your mind and you alone are it's creator!

However, this good side has a corresponding negative side (notice I didn't say "bad" side, just a negative side). There is a tendency among programmers to "just start coding" and the program then flows from the keyboard. Sometimes that's exactly what happens, the program flows from the keyboard and not from the mind. It's like a writer who gets lost in his words, or like a builder who never develops a detailed plan of a building before he starts building it — it just ain't gonna work!

There's no reason to feel that a program needs the same kind of spontaneity that the idea itself had. A program that is worth writing, is worth writing well (to coin a phrase). And, if it's worth writing well, it's definitly worth thinking out first. If you pick up a program and look at it, it's very easy to see how much thought went into the development of it. One easy give-away is the line numbers. Are they incremented by the same amount throughout (with the possible exception of the ending line number of each routine)? Does the code flow smoothly or does there seem to be places where the GOTOs just don't seem to fit, places that look like they've been changed several times? Does the physical appearance of the program seem

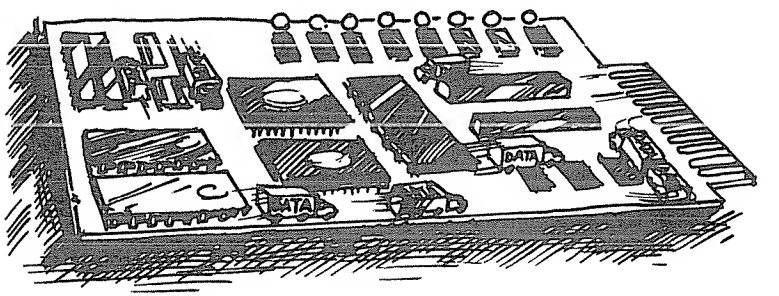
neat and clear?

All of these, and more, go into making a really top-notch program. There's so much to consider, that in a book of this scope, we really can't get into all of the factors that affect a program. For those of you using BASIC as a regular language, I strongly recommend "*The Little Book of BASIC Style*" and, for all programmers, "*The Elements of Programming Style*" both of which are referenced in the bibliography in the back of this book.

Now it's time to press on. In the next chapter we will be addressing the idea of a program overview and how it might fit into the scheme of program development. As we do, I'd like you to continue to think about the material we have looked at in this chapter to see how it can be applied. We will not be reviewing this material and, so, we will assume that you have fully grasped the significance of it before proceeding. This material is important and should be carried forward as the discussion proceeds.

CHAPTER 5

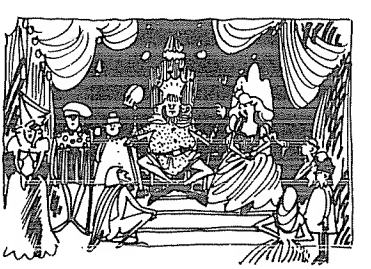
Program Overview



What on earth is a program overview? One possible definition of an overview is a look down on, or a description of, something. An overview for a program is usually, but not always, a written description of the program, its functions and data requirements (both input and output). For our developmental purposes the overview should be anywhere from a few notes to several pages of description.

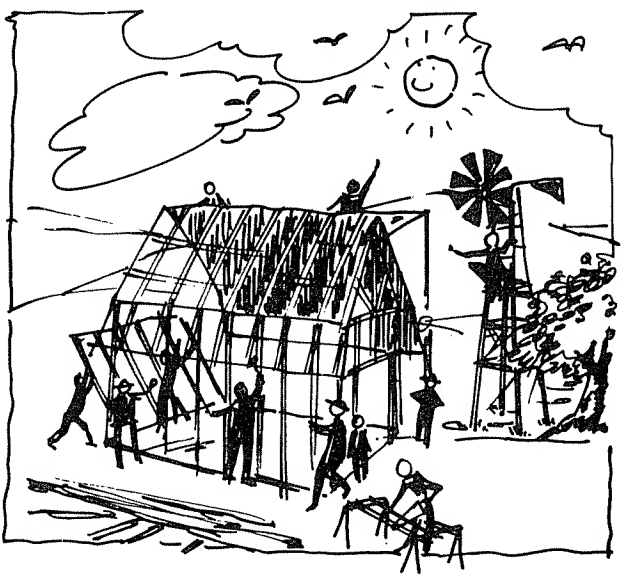
The overview should contain, at a minimum, a line about each functional block of code (routine, subroutine, function statements, data statements, and data definitions), a description of the input data as well as a statement about the output requirements. There should also be a description of the flow of data through the program, with any tricky logical decisions laid out in a decision table fashion. This need not by really neat, but should be of sufficient clarity to aid you in the various developmental tasks you will be undertaking.

The overview can be done in an outline format or whatever format you personally prefer. I like to use the outline format because



it is inherently hierarchical, and the programs that are developed under this scheme will show the clarity of this hierarchy. By the way, the term hierarchical means a structure that has rows or ranks with differing levels of importance. In the case of a program, a major routine may be built up of one or more minor routines. Thus there is an inherent hierarchy or level of routines. This is a common term in the data processing world and one with which we should all become more familiar.

As we have done in the past, we will discuss the reasons for the importance of the program overview along with some simple ways of building the overview. This will be covered especially in the process of the development of the overview of the program that we will be developing during the remainder of this book.



Let us begin with a very simple example, one that will actually take longer to develop the overview than it will to write the program (which will be left as an exercise for the reader). Let's go back to the trivial example of the last chapter, building a program that will calculate the area of a rectangle. As we are all aware, this is a program that the world could probably live without, and so we won't dwell on the program so much as the developmental techniques themselves. The program can be considered merely a means of providing us with the framework for our example.

The overview, then, will reflect the need for getting and verifying the input, the calculations, and the output routines. We will include the various "overhead" routines, those used to advise the user of the program function, as well as the clean-up routines used at the beginning and end of the program. Such an overview might look like the following:

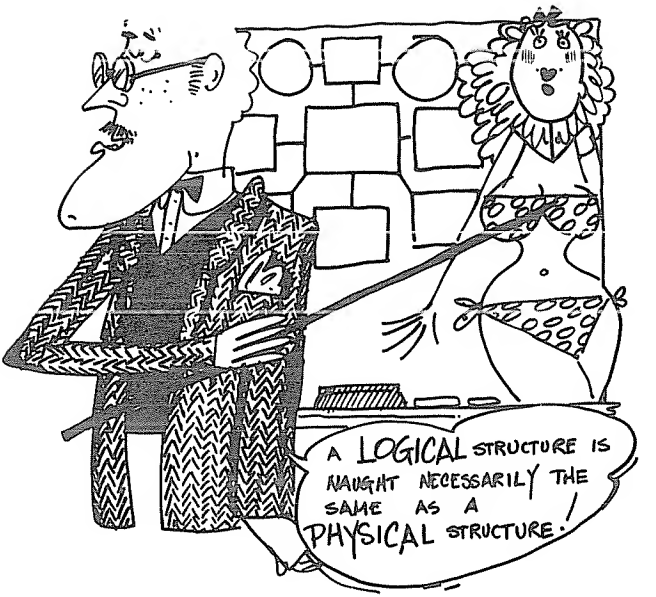
```
I) Program to Calculate the Area of a
   Rectangle
   A) Initialization Routines.
      i)   Clear the screen
      ii)  Display the program title
      iii) Initialize and define all
           variables
   B) Prompt for Input
   C) Input processing
      i)   Read input variables
      ii)  If End of Data (Length) and
           Width) terminate
      iii) Validate variables
           a) Length or Width equal to
              zero? If so, error.
           b) Length or Width less than
              zero? If so, error.
   D) Calculate Area Width x Length
   E) Display results
      i)   Print "AREA = A
      ii ) If Length = Width print "THIS
           IS A SQUARE"
      iii) Go get more input
   F) Termination Routine
      i)   Print termination message
      ii)  Terminate
```

As you can see, this overview took longer than it would take to write the program that corresponds to it. But because the program is so simple, it makes it easy to visualize the entire program and see how the overview lays the program open for inspection. Also, it becomes relatively simple to write the program from this overview.

We will be looking at flowcharts a little later on and you should be able to see the similarity between the overview and the flowchart

for the same program. We will continue to use this very simple program example for looking at these various techniques because we can quickly evaluate the effectiveness of the new developmental method under discussion. Of course, I hope you don't think that this sort of care should be taken for every program. Some programs are so trivial that they can be designed at the keyboard, but many are not. Part of learning how to program well is the ability to recognize which programs are which!

In the meantime, let's consider some of the important reasons for taking the time to develop a program overview. If we assume that an overview is to be used as part of the process of developing a program rather than as the only developmental step, we can usually spend only as much time as necessary on it. On the other hand, if we use only the overview, we should spend a considerable amount of time developing it. The purpose of the overview is to allow several things to happen: (1) help us clarify our thinking about a program, (2) assist us in determining whether or not we have defined all of the parts of the program that are necessary, and (3) form part of the final program documentation. But the most important function of the overview is to ensure the logical structuring of a program.



This structuring is not necessarily a physical structuring, but rather is used to assist in logical structures. The primary difference is that the logical structure requires that certain functions be performed, but these functions do not have to have their code physically adjacent. In fact, based upon the principles we have already discussed, it makes more sense to have the high-level routines in one place in a program with the lower-level routines in another place. This makes the more usual "working" code smaller and easier to read.

A "rule of thumb" that is often used states that a given routine should not occupy more than one page. This keeps the amount of code to a minimum and makes it very easy for someone to pick up the program, quickly grasping the significance of a given block of code. If you've ever looked at a program where the code was scattered all over the place, with no spaces anywhere, you know how difficult it can be to read. Imagine the difficulty you might have trying to maintain that program! Now, before you throw this book in the trash, hold on a second. I know that it's popular to remove all spaces from a program to minimize the time it takes the BASIC interpreter to scan a given line, as well as to conserve memory. This same reason is given for packing as many statements on a single line as possible. If this is necessary, by all means go ahead and do it. But, you might want to consider keeping a copy of the program around that has spaces, remarks, and single statement lines. This can be used for maintenance purposes and for storage with the program's original documentation. In fact, I usually recommend not "scrunching" programs unless it is absolutely the *only* way to get a program to run. The difference in speed and memory utilization isn't usually all that great anyway!

While we're talking about the physical appearance of the program, let me drop a few other ideas into the discussion. There are a few easy tricks that will help keep program structure obvious. One of the easiest tricks is to simply use indentation. For example:

```
100 FOR I=1 TO 100
110       PRINT I;
120       J=SQR(I)
130       PRINT J;
140       K=I↑2;
150       PRINT K
160       FOR L = 1 TO 10
170          M(L)=0
180       NEXT L
190 NEXT I
```

This makes it very easy to see the related FOR...NEXT pairs. The amount of indentation keeps the FOR and the NEXT at the same point on a line, while the indented material is obviously part of the corresponding FOR...NEXT loop. This sort of physical structuring

not only allows a neater program from a visual standpoint, it also shows the thought put into the physical structuring. By the way, this is another example of a hierarchical structure.

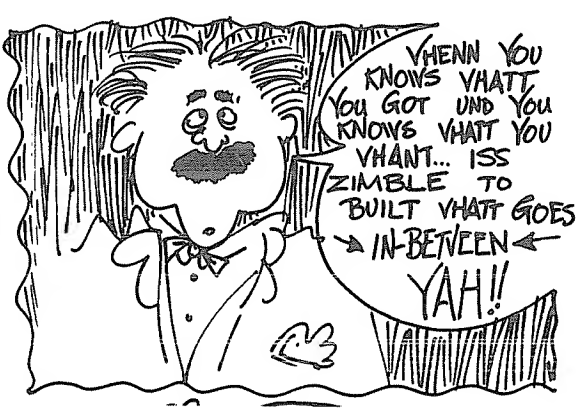
Since we are talking about ways to develop a program, it seems reasonable to also talk about ways to visualize a program. The overview of the program is just that, a way to assist you in visualizing the program.

"But," says you, "how do I develop an overview to visualize the program if I can't visualize the overview?"

That's a good question, and we'll answer it right now. Normally when we are about to write a program, we have an idea of the data that goes into the program as well as what we want to get out of the program. The first two blocks of the overview are already done for us! This might look like:

```
I) Program to .......
   A) Program Input
      1) Get input data in format ....
      2) Validate data using ....
.
.
.
   Z) Program Output.
      1) Produce reports....
      2) Generate output file....
```

As you can see, there's not a whole lot of information that is contained in this overview. It's not really much more than a beginning and an end. But, therein lies the secret — it's a beginning!



With the definition of the input and its editing requirements as well as the definition of the output and its various reports and files, you are in a fine position to define the middle blocks of the overview. Knowing what you have, and what you need, it should be a simple matter to define the processing necessary to connect the two.

As is usually the case, the process of arriving at a real definition of this processing is often an iterrative or repeating process. We will often begin to define these requirements then discover that we've either forgotten something or we've got more processing than is necesary. As we go along we can correct this and the final result should be a description of the actual processing requirements.

To assist you in understanding the material that we are covering, let's go ahead and plunge right into the overview for the check writing program. We'll begin with the definition of the input and output requirements. Why are we starting there? That's an obvious question and, fortunately, there's an easy, obvious answer. We're starting there because if we don't have known output requirements, there's no need for the program ('Cuz we don't known what the program's supposed to do!). If we don't known what input we need to get the required output, we have one of two problems. One is that we don't really understand the function the program is going to solve, and the other is that we don't have the foggiest idea of what it is we're doing! If we really don't understand the input/output requirements, then we need to go back to the requirements stage where we define what we are going to do. That should be followed by the definition stage in which we define the approach to be taken. Once these two steps have been completed we can then continue to the design stage.

Since we understand the function we are performing (the problem we are to solve) we can press on. Our input will consist of a payee (the person or firm the check is being written to), the amount that the check is being written for, and possibly a note to allow us to specify why we are writing the check. We will also need to have a date on the check, and possibly we will want to keep track of the check number to make sure that the program knows about all the checks that have been written.

"Sound like we've covered all of the input/output requirements?", I ask.

"Not by a long shot, monkey ears!"

Right, we need to discuss the other "hidden" inputs and outputs.

For example, all of this data is being stored in a file that keeps track of the checks (we'll call it a Check Register File, or CRF for short). Anything else? Well, if we have more than one account that is being handled by this program, it might be nice to have a separate CRF for each. Look at that, will ya! Already I've shown my propensity for (means predilection for, or inclination to) developing acronyms which are a sort of nick-name. That's perhaps the bane of programmers. We often come up with names that are longer than are allowed by the computer. For example, the file could be called the CHECK.REGISTER.FILE on a large IBM computer (System 360/370 or 303x series) but not on most microcomputers, thus the need for the acronym.

Anyway, are there any other inputs or outputs that we need to consider? The answer may or may not be yes at this point. See how positive I am? The reason for providing an answer of the "definitely maybe" category is that although the program we are developing may not have any other files to consider, we may be interfacing this program to other, pre-existing programs that have specific file requirements. It is necessary that we consider those programs as we design the current program so that the files will interface correctly. By the way, we've used two terms synonomously here, files and input/output requirements. A file can be either a data file on cassette tape, floppy or hard disk, or it can be a printed report. Usually a displayed piece of data is not considered in the category of input/output for the purposes of this part of the design. We'll get to that in a moment.

As long as we're talking about files, let's consider one other important part of the design. Where will the file reside? This is, perhaps, a moot point. If your system only supports cassette files, or only disk files, then the question's already been answered! For most users that have disk files the question is answered because of the speed difference between disk and tape. But, there may be some particular reason for supporting say, a stringy floppy as opposed to a disk.

This consideration, and others like it, are the things that you should be considering as the design of the program evolves. I hope that you noticed the key word in the last sentence: evolves. That is precisely the way in which a program design should come about. Beware of any program design that seems to spring into mind full-blown. Again: small, simple programs are excluded from this dis-

cussion.

Program development is an evolutionary process. In fact, some large companies are already experimenting with a formal version of the process we in the microcomputer arena have used for a long time — iterative design. For us it's usually a matter of necessity; the original design didn't work, so we just keep on playing with the code until it does work. For these large companies it has become a way of getting better user involvement.

That term "user" is another very important word. For programs developed for our own personal use, we are the user. Since that is the case, getting user involvement from the very start is guaranteed (unless, of course, we develop the program in our sleep). But, what if we are getting together with a few friends to develop a program that will be shared between several other people. Who should be involved? What should the amount of involvement be?

The answer to both questions is easy to state and, sometimes, very difficult to put into practice. The user (frequently called the "end user") should be involved from the beginning of the program design (if possible) and should be as involved as possible. Obviously this excludes any products that are being developed to be marketed. For this sort of program, the end user involvement can be replaced by carefully examining the requirements for the program being developed, and using friends as "model" end users.

Let's get back to one of our defered subjects (displayed data). Out of all of the various program development topics covered in the trade journals over the last several years, one very central point has been raised. That point is that the program must be "user friendly."

Just what does that mean? To begin with, the program should communicate with the user in terms the user is familiar with, not programming terms he may or may not be familiar with. Secondly, the program should never, never assume anything.

A common error seen in many programs is the use of a YES/NO question to get a response from a user. Such a question may ask if the user wants a report printed. Nothing wrong with that! It uses terms the user knows and the response is something from everyday usage. Right? But in the code, there is a check for, perhaps, "Y" and "YES" with any other response being considered as a negative response. But, what if the user typed in "YEA" because his finger slipped and got the "a" key instead of the "s" key? Sorry about that, no report! The program made an assumption that since it wasn't one of the expected positive responses it must have been a negative response. Always check for all legal possibilities and reject any that aren't logical. It's always better to ask again than to guess.

Displays should be designed to speak clearly to the user. The Chinese have a saying that a picture is worth a thousand words.

The various displays, especially menu displays, should speak volumes. Some simple rules that might make this easy to do are presented for your enjoyment. First, make all screens in a given program consistent, both in format and in usage. Second, make any letter or number codes used for function selection consistent. For example, don't use "E" to mean EDIT in one screen while "E" means END in another screen. Also, while we're on this topic, try not to use numbers if at all possible. Numbers are great for computers but people don't think in terms of numbers.

Below is a sample screen that shows some of the ideas we've discussed as implemented. The screen is from a mailing list program and allows the selection of several different options. This is the master menu.

```
*********************************************
*------------- MAIL LIST MENU -------------*
*ENTER SELECTION ==>                 05/25/82*
*                                    11:15:21*
*     1 ADD          ADD A NEW RECORD        *
*     2 DELETE       DELETE A RECORD         *
*     3 UPDATE       CHANGE A RECORD         *
*     4 LABEL        PRINT MAILING LABELS    *
*     5 LIST         PRINT MASTER LIST       *
*     6 LIABILITY    PRINT LIABILITY REPORT  *
*     7 RENEW        RUN AUTO-RENEWAL        *
*     8 TERMINATE    RETURN TO DOS           *
*                                            *
*                                            *
*                                            *
*                                            *
*                                            *
*********************************************
```

This program was designed for a computer club to maintain the mailing list and to perform some special routines used for the newsletter. The display allowed the use of a number for selection, but only with the support of the short title (such as ADD or DELETE) as well as having a descriptive entry on each function line. This allowed the user to select an option knowing what would be performed. Obviously, the RENEW and LIABILITY options were a little strange until you understood the function of the program, as well as the user of the program.

The key here is that you can make the screen tell the user everything needed to make the selection without undue reference to the program documentation. Recently, the large computer world has been looking at the way we in the micro world have been able to market programs with little documentation. That has been accomplished by having user friendly programs. Programs that help the user to run it. One of the keys to this is having complete selection screens. Screens that are not ambiguous or confusing. Screens that will *always* look the same so that the user will be able to look at any screen in the program and be able to understand it. The hardest part is to understand that there are users out there that are not computer oriented. They may own computers and use them, but

they are not oriented toward them and don't want to learn any more than is necessary to use the computer to perform the functions they want it to perform.

This is not unreasonable and is perfectly understandable. But what does it mean for us, the programmers? It means that when we design a program and are considering the various routines and functions needed, that we have done so with the end result in mind. That we have carefully designed the program using tools like the program overview or like flowcharts and pseudo-code (the subjects of the next Section).

So far we have only begun the design process — the "leg work" if you will. We still need to perform the detailed design, the nitty-gritty of programming. I'm aware that you might want to know why, but there's so many reasons that I'm going to answer those questions as we go along. We will plunge directly into the next Section now, but we'll take along the ideas we've already covered. You'll see that they will come in handy.

Section 3 — Designing the Program

In this section we will take a look at what it takes to go from our preliminary design of the last Section to a final program design that is ready for coding. We'll be using two of the more common tools of mainframe programmers (flowcharts and pseudo-code). It isn't necessary that any program utilize both of these techniques. We're looking at them both so that you can chose one or the other for your own personal use.

Taking a program from the preliminary design is not an extremely difficult task, but it does require a great deal of attention to detail. Once this step is completed, you should be ready to directly translate the final product into a running program. Assuming that the translation effort is done properly, there should be a minimum of testing and debugging required.

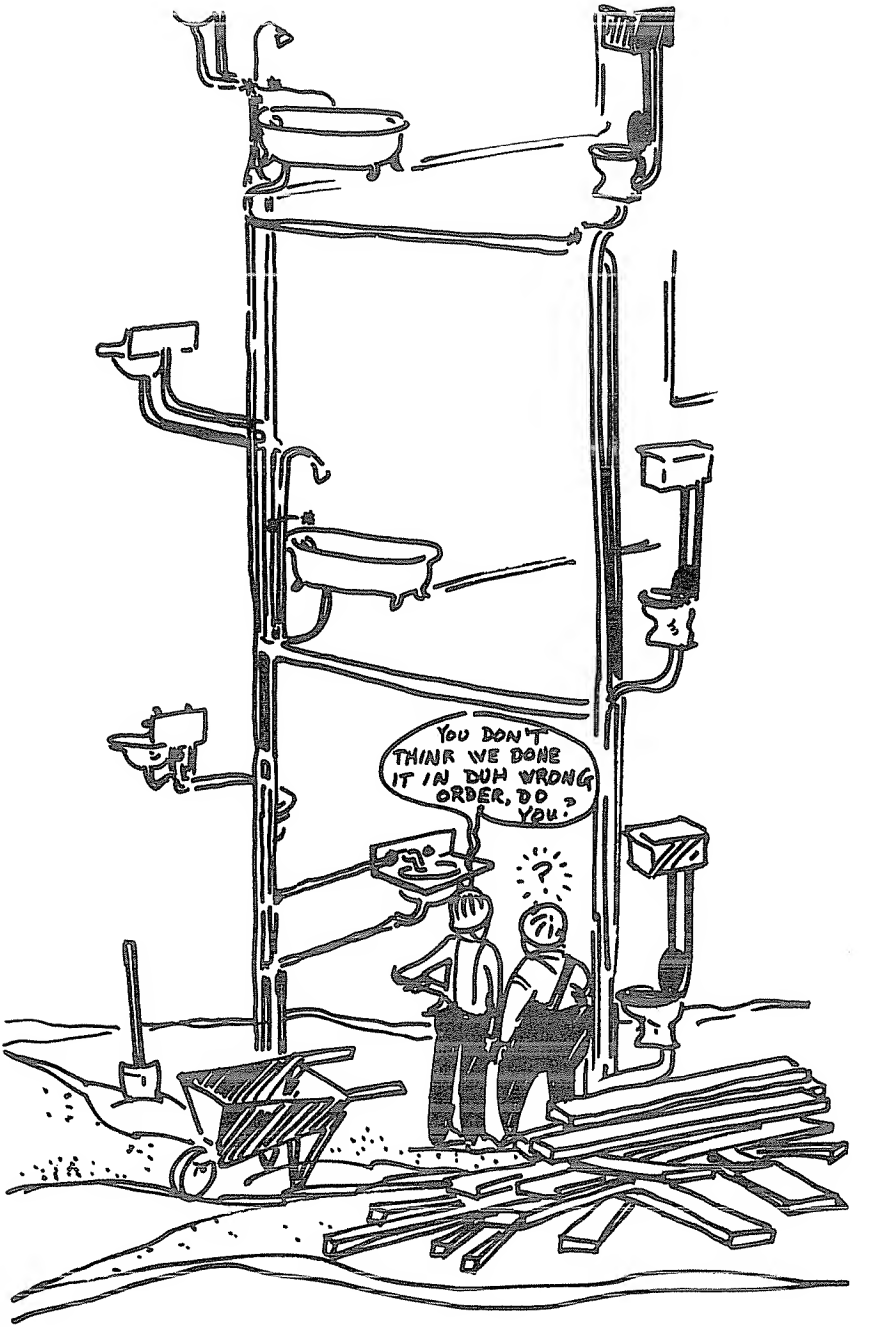
The primary emphasis here will be on the unique requirements of each technique for the final development. We'll also take a look at the items which must be considered in the final stages of program design. Again, the end result is a method of rapidly converting any intermediate idea into a working program.

CHAPTER 6

Flowcharts

As we have already discussed, the development of a program can be likened to the design of a building. There is a large amount of planning that must take place. The design of the various subroutines and functions is similar in concept to the design of the plumbing and electrical requirements for a building. Just as a contractor uses a blueprint to map out the development of a building, so too the programmer has a similar tool. The tool is called a flowchart.

The flowchart is not a new development, and in fact goes back to the very early days of program design and development. A flow chart may be used in two different ways. The most common method is the mapping of the flow of code — the sequence of instruction execution. A second technique, developed by IBM, is called "HIPO" (Hierarchy plus Input, Process, Ouput). This technique shows the flow of data through the program rather than the execution sequence


YOU DON'T THINK WE DONE IT IN DUH WRONG ORDER, DO YOU?

of the instructions. Although this may be called a flowchart, it's usually called a HIPO Diagram or HIPO Chart.

We will not get into the HIPO Diagram too much, but will touch lightly on it near the end of the chapter. For the most part, we will concern ourselves with the use of flowcharts for program design.

To begin the discussion let's focus immediately on what is conveyed in a flowchart. To do that we need to understand the symbols used in a flowchart. There is a specific number of symbols that are approved by the American National Standards Institute (ANSI) and these are almost the only symbols we will be using. We will be using ANSI X3.5-1970 standards which conform to ISO (International Organization for Standards) Standard 1028. Naturally, various flowcharting template manufacturers have added their own special symbols, but we will not use them. The application of some of the symbols will be, admittedly, my own preferance rather than a "standard." This will only be in those cases where there is no standard.

Figure 6-1 shows the full set of symbols and the keyword associated with each symbol. Since this may be the first exposure for some of us to this material, we'll take a few minutes to go over the meaning of each of the symbols to make sure that we're all in "synch." Please, for those who are familiar with this material, stick with us. That way we'll all have the same understanding of the material as we go forward.

The PROCESS symbol (a plain rectangle) is used to indicate normal processing. That is, processing that does not include I/O, decision making, or transfer of control (GOTO). It may be used to indicate a GOSUB, but there are better ways of doing that.

The INPUT/OUTPUT symbol (a parallelogram) is used to indicate the transfer of data from a file (disk, cassette, or stringy floppy). This is not used for printed material, data that is displayed on the video unit, or for material that is entered at the keyboard in response to a prompt; there are special symbols used for that.

The CONNECTOR symbol (a circle) is used to show control transfer from one part of the program to another, but only if it can be done on the *same page* (of paper). If the flowchart is big enough to require multiple pieces of paper, use the special OFFPAGE CONNECTOR (a pentagon) to indicate control transfer to a point on a different piece of paper. Please note that this OFFPAGE CONNECTOR is not an ANSI standard, but is rather one added by IBM which I like because of the positive statement that it makes. This is due to

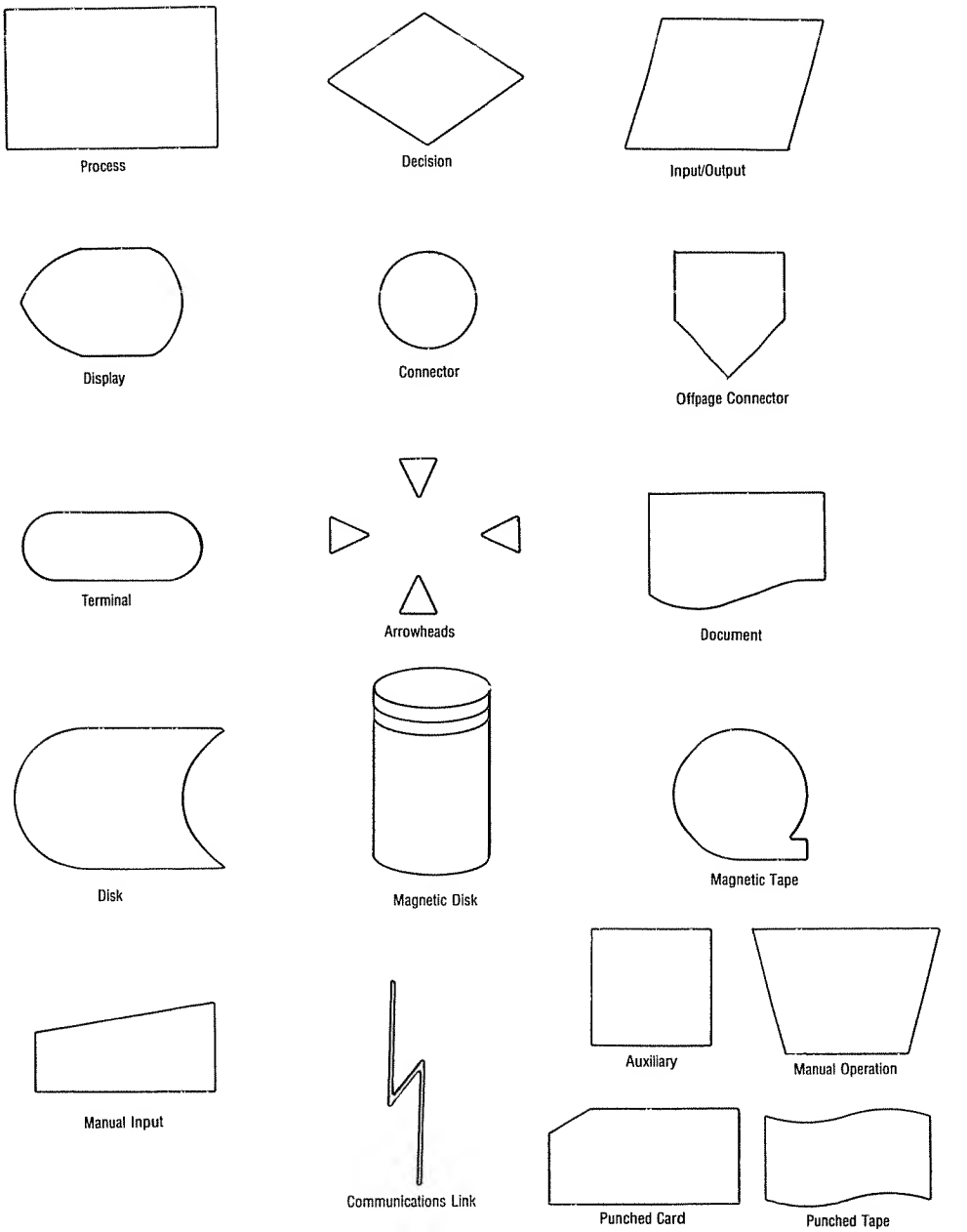


Figure 6-1. ANSI Flowcharting Symbols

the fact that a flowchart should clarify, not confuse!

COMMENT symbols are part of a group of symbols called "composite symbols," formed by changing one of the basic symbols. In this case the PROCESS symbol is changed by removing the right side line, and by running a dotted line to the symbol from the appropriate point in the flowchart.

ARROWHEADS are used to indicate the direction of processing flow. This is the instruction sequencing, not the data flow! ARROWHEADS are placed on flowlines (the lines that connect the various flowcharting symbols). These lines, like electrical schematic drawings, may cross without implying a connection. Connections are positively indicated (see the sample in Figure 6-2).

DECISION blocks are indicated with the diamond shape. This is used for IF statements. The normal flow is into the top of the diamond, and there are three possible exit conditions (less than, greater than, and equal). These are, of course, not always used in this fashion. We'll see some specialized usage of this block as we discuss STRUCTURED PROGRAMMING.

The TERMINAL symbol does NOT relate to a Video Terminal. Instead, it relates to a terminal or exit condition in a program or routine. It is represented by a capsule shape, a sort of flat oval like a race track. This is used to show that the logical end of that function has been reached. It may mean the end of the program or merely the end of the routine.

There are other symbols that are commonly used to represent the outside world. These are the so-called "system" symbols. We will be concerned with six of them. These relate to various storage media, as well as input/output devices (as opposed to files).

Our first symbol relates to printed reports. The DOCUMENT symbol is merely the representation of a piece of paper. As usual, the purpose is to make clear, at a glance, the function intended. Continuing that trend, the symbol for a magnetic tape (cassette tape) is a circle with a line under it. This is a composite symbol that usually refers to a reel of tape on a large computer, one that is similar to a reel-to-reel tape recorder. Since that's the only symbol available, we'll go ahead and use it.

Disk storage has two different symbols associated with it. One is a general "on-line storage" symbol, and looks like Buck Rogers 1930's era space ship. The other symbol looks like a 55 gallon drum.

For our purpose, we'll use the spaceship symbol. It seems to be more universally used, and it is not a composite symbol which the disk symbol is. I've shown both in Figure 6-1, but have indicated a preferance for the former.

Since there isn't really a symbol that is appropriate for the stringy floppy, we'll consider it what it really is, a tape drive, and use the TAPE symbol for it. It really doesn't matter unless the program is device dependent; then, it would probably make sense to add the word "STRINGY" to the graphic representation, or to add a comment block to that effect.

Manual input (data input from the keyboard) is indicated by the sort of squished rectangle. This is also used if you have switches that can be read by the program. Any form of input that does not come from a file (disk, cassette, or stringy floppy) should come from this manual source or from a communications source. A COMMUNICATION LINK is indicated with the lightning bolt. This represents data either transmitted from or to the computer.

The final symbol that we will be working with on a regular basis is the DISPLAY symbol. This is used to indicate data being displayed on the video unit. The symbol is shaped like a simple CRT unit, with a curved face and a strangly tapered tube. Again, the ability to recognize it quickly was the more important design consideration.

Figure 6-1 contains the remainder of the symbols that are used in flowcharts simply because we need to be complete. We will not be using any of these symbols as we proceed through our discussion because they are related more to the development of software systems than to the development of programs. We will touch lightly on the considerations of program design that will lead to good system design, but that will be incidental rather than as the primary thrust of this Section.

We will begin our discussion of flowcharts with some simple guidelines regarding the use of flowcharts. Since the time of Alexander the Great and The Great Grape Ape, one question has stood out in the history of man. How detailed should a flowchart be? The question has a very simple answer, and a moderately simple answer. We'll cover each in turn.

The simple answer says that the flowchart need only contain material for each routine or major logic junction.

"And," the astute student asks, "just what the heck is a 'logic junction'?"

Funny you should ask, I was just getting to that. A logic junction is any point in a program where a change of instruction sequence will take place as the result of a decision made based upon data. In other words, as the result of an IF or, perhaps, an ON...GOTO or ON...GOSUB. These are usually flaged so that the programmer can make sure he keeps the decision in the program where needed.

The moderately simple answer that I mentioned says that a flowchart should maintain enough detail to allow you to code the program without any unnecessary references to other documents as far as the logic of the program is concerned. Of course, it may be necessary to refer to a print layout to ensure getting the report correctly formatted, or to a file layout to make sure the files being processed by the program are correctly defined. The flowchart is not used for those functions, it's only for logical structuring of a program.

So far, this hasn't really told you anything has it? Well, to be frank, there's not really much to the great mystique of flowcharts. They can be very detailed or very skimpy. For example, the flowchart below is perhaps as short as any flowchart could ever be. On the other hand, a flowchart can have every possible line of code on it. That's overkill and isn't recommended.

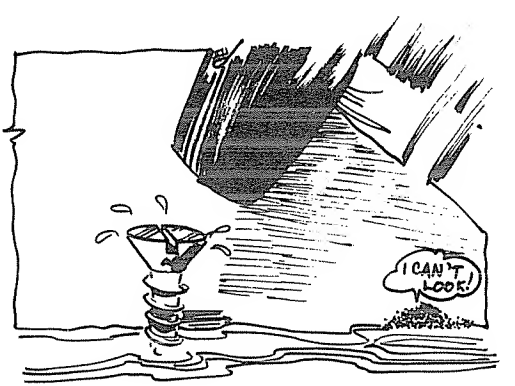
Later on in this book we're going to be talking about how to document a program and the flowchart will become a permanent part of that document. Assuming that you will want to maintain the program documentation, it makes little sense to force a flowchart to be so detailed that every little change to a program will require redrawing the flowchart. That's only one of the reasons that a flow-

chart should not be a line-by-line representation of the program. There are many other good reasons, most of which have to do with the so-called law of diminishing returns. That is, beyond a certain point the amount of effort put into a flowchart exceeds the possible benefit of that effort.

For example, if you have defined a program sufficiently well enough to be flowcharted, you've probably already spent a lot of time thinking about the program itself. Once the preliminary design is done, the flowchart should be only enough to solidify that design. Just enough to make sure that you can remember where you are going with the program. Spending the necessary amount of time that it would take to build a super-detailed flowchart will require an inordinate amount of attention to detail at a time when you should be concentrating on merely noting the functions, at a moderate level of detail, that need to be performed. This can be done more rapidly and thus there is less of a chance that you will omit some high level detail. The old saying about not seeing the forest for the trees really applies here.

The bottom line, then, is that there is a limit to the usefullness of flowcharts, and that limit is one placed upon the *purpose* of the flowchart. Again, we're not trying to limit your creativity. In fact, we're trying to do just the opposite! By allowing you to place only the amount of material in any of these design techniques as you will need, you alone are defining how these tools will be used. You are the final judge of the effectivness of any tool that you are going to use.

Some books I've read have referred to the misuse of various tools by comparing it to using a pair of pliers as a hammer, useable



but certainly not ideal. Actually, it's that very "usableness" that we're trying to convey. Who cares if it's a misuse as long as it works? One of the things that I push very strongly for is the basic right of every programmer to be an individual and to develop programs in the means best suited to his own disposition. This does not mean that I'm against egoless programming in a commercial environment where several programmers may be working on the same program. Just the opposite! I'm firmly behind the efforts to make all programming egoless, that is, free from the nagging need to make each program a personal statement. If a program is being developed by one programmer for his own use, he can do whatever he wants to do. All we're trying to do here is to set the stage for efficient development of clean programs.

To see how we could use flowcharts, let's take a look at two examples of flowcharts for the program to calculate areas (our simple program from the last chapter). This first flowchart (Figure 6-2) shows a moderate level of detail, sufficient as a map or guide to writing the program. Notice that the decision blocks are the only places where there is great detail. This is as it should be. Regardless of the amount of detail in the overall flowchart, decisions should *always* be clearly spelled out so that when the program is written later, there will not be any hidden logic that would take extra time to reconstruct, or that might possibly be forgotten.

The second example (Figure 6-3) is a fully detailed flowchart, drawn at the instruction detail level. In this case, we can see that the program has already been written. The flowchart merely needs to be translated on a symbol-by-symbol basis to have the entire program ready to run. Of course we already know that this is a bad practice. To reiterate, we don't develop a program full blown, we do it step by step so as to ensure the accuracy of the final product. If we are doing the last step as one of the intermediate steps we may, and quite likely will, miss some important detail. That's the primary reason for showing this flowchart at this time. There's no need to ever develop a program flowchart at this level of detail. I realize that one should never say never, but in this case it's a reasonable thing to say.

As you can see from both of these examples, however, there is very little left that is required as far as coding the program is concerned. The structure of the program has become apparent due to the flowchart; the mainline comes first and is treated as a fall-

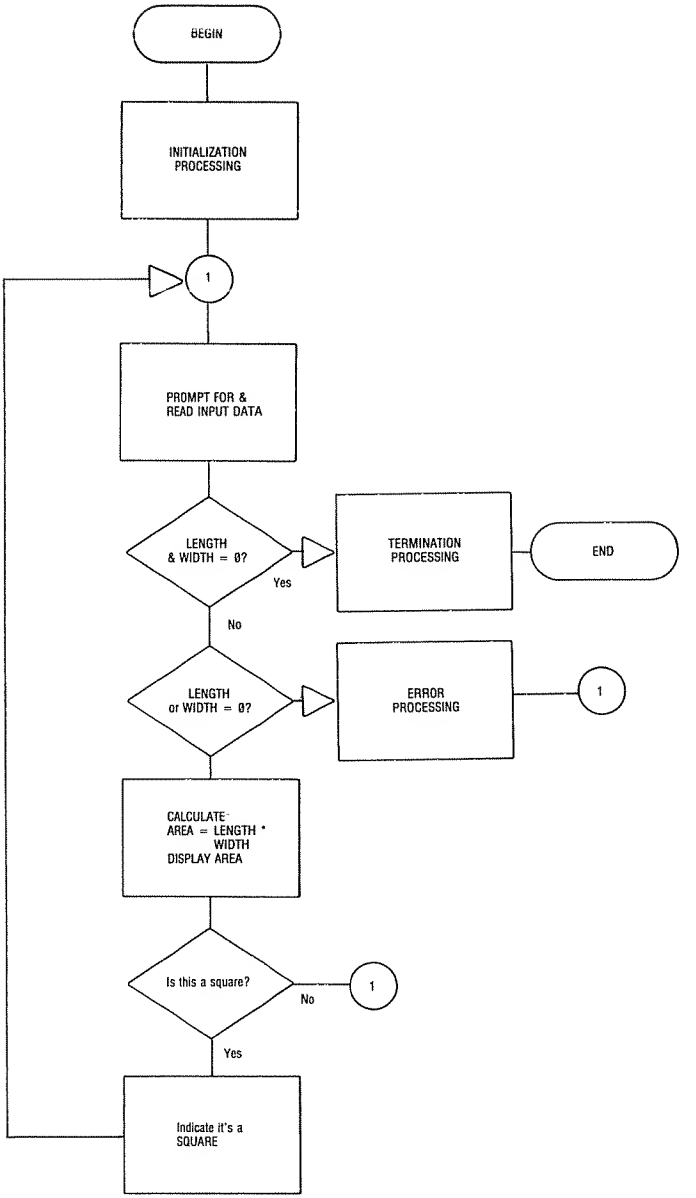


Figure 6-2. Flowchart for AREA Calculation Program

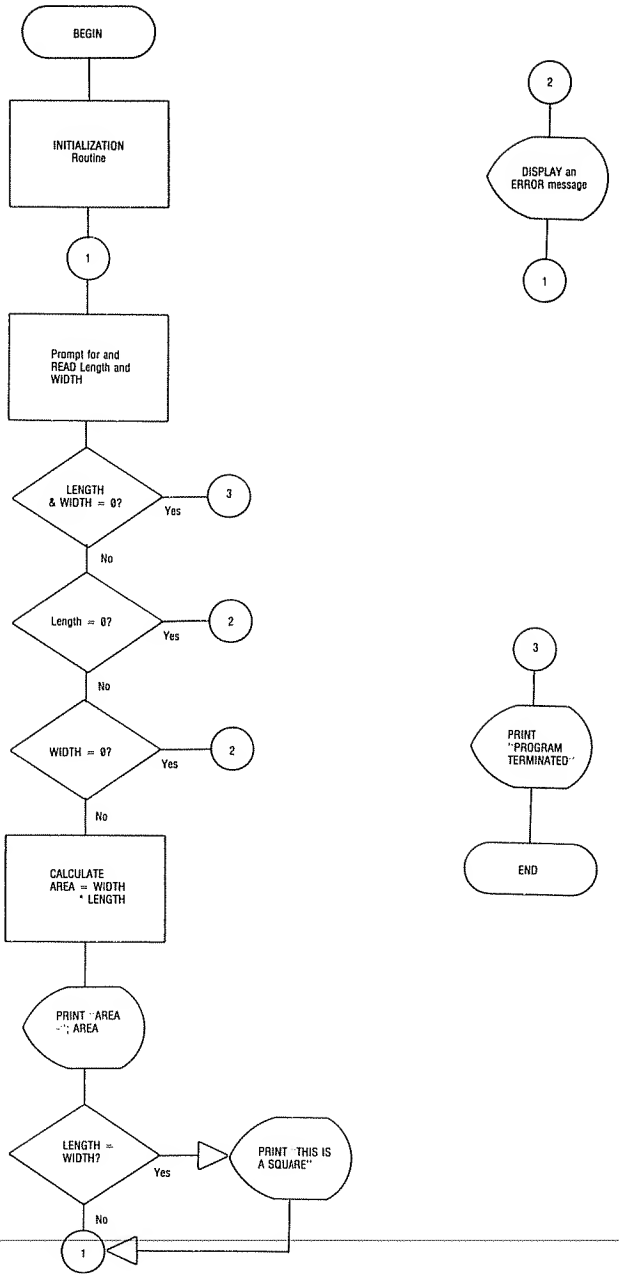


Figure 6-3 Full Detail AREA Calculation Flowchart

through structure. All subroutines are last and are entered only as necessary. The only branches out of the mainline are to return to the prompt or as the result of an unusual condition; either an error or a termination.

This is *not* structured programming, however. For those of you who are interested in structured techniques, the third flowchart (Figure 6-4) shows the structure as it might appear if we were to use this technique.

By the way, as long as we're touching on structured techniques, there are some flowchart related functions that should be mentioned. Figure 6-5 shows some special structures that are used in flowcharts that are based upon a structured development technique. These are really just specialized uses of the symbols already discussed.

"Ok, just how does one start to draw a flowchart?"

Another pertinent question! If we stop and think about it, we have one of those simple mechanical tricks that we can use to build a flowchart starting point. We begin with a terminal symbol labeled "BEGIN." Having begun, we simply follow through by adding blocks that describe the various processes that the program must perform.

These additional blocks should have a structure that is similar to that of the program overview. That is, we should be building upon the developmental work we've already done. The flowchart is not, or at least shouldn't be, the first stage of program development. Assuming that it isn't, we can see that we must already have an idea of what the program is to do as well as how it is to do it. With this information, we can then break the functions into logical structures.

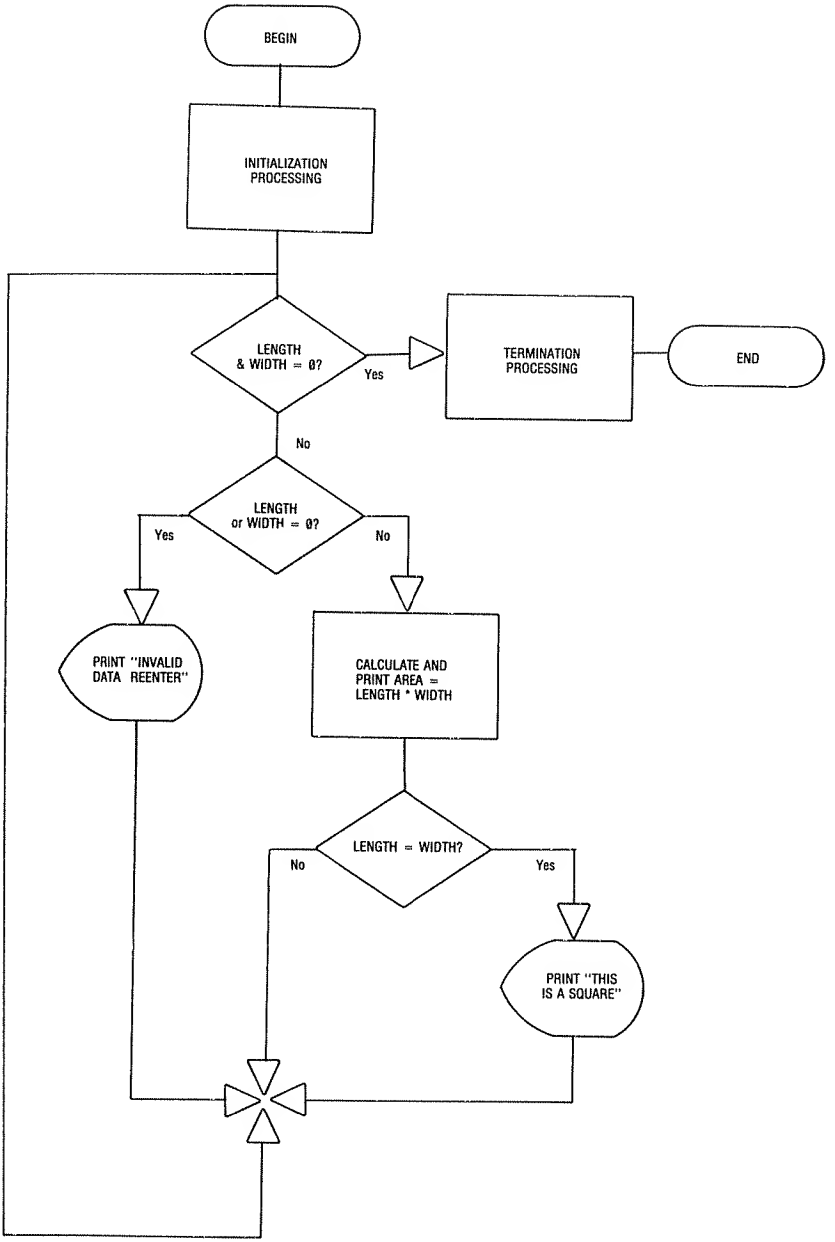


Figure 6-4. Structured Flowchart of AREA Calculation Program

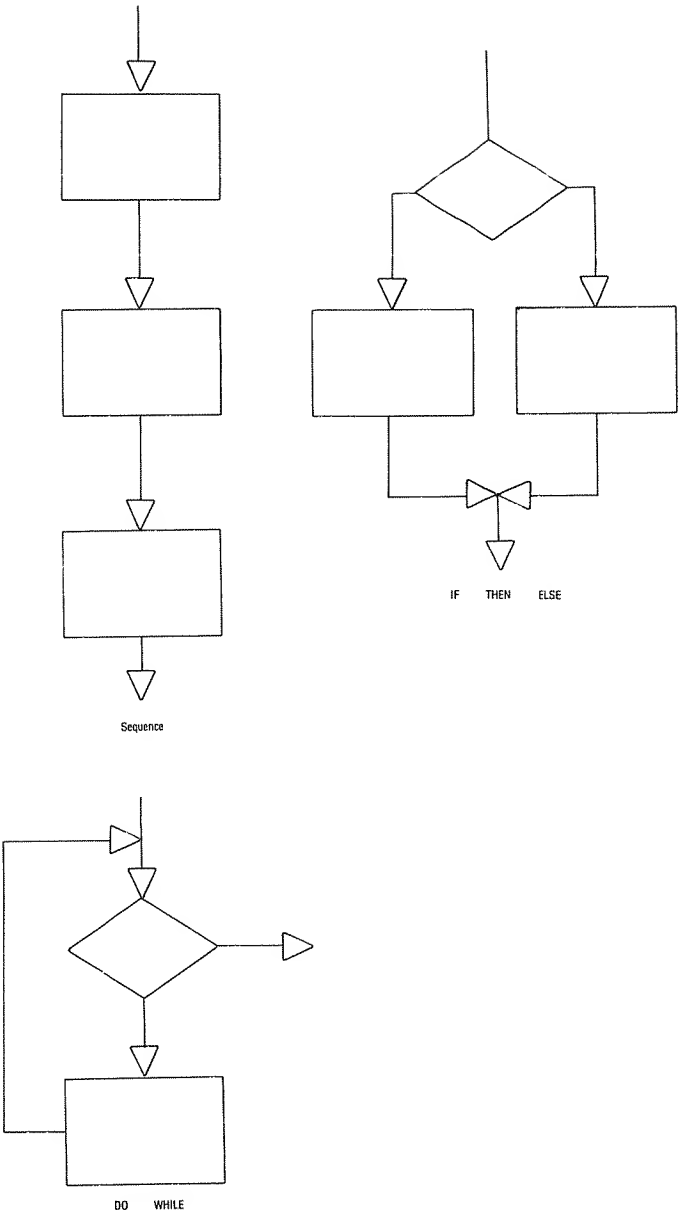
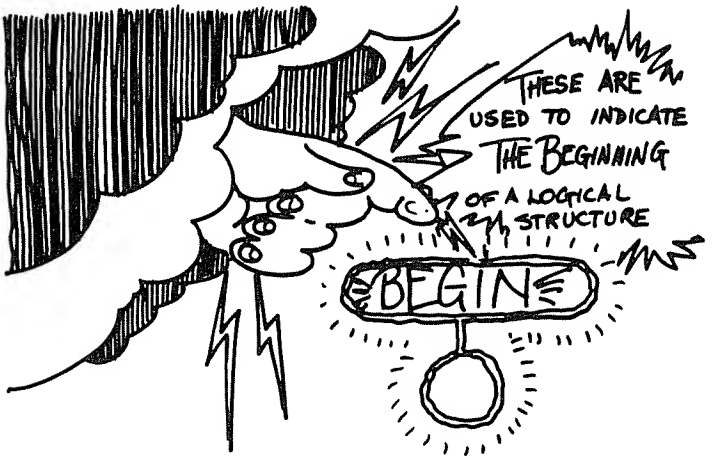


Figure 6-5. Structured Programming Flowchart Symbols



"How does one determine the beginning of a logical structure?"

That's another good question. The answer is, fortunately, very easy to express. As we think of the various parts of the program we see that there are places that we must "jump to" in order to perform either the repetitive or the unique parts of the program. That is, we may be "looping" through some routine (or many routines). The entry point to that loop is the beginning of a logical structure. If we branch out of the main code to perform some specialized section of code, we have another logical structure that begins at the point where we enter the particular routine.

Does that make it all clear? I certainly hope so. If it doesn't, don't worry about it too much; we'll be taking a closer look as we go through the flowchart for our main program.

Another question that's often heard has to do with the desirability of just building a "block level" flowchart. Such a chart is shown in Figure 6-6 for our "area" program. There isn't much that is different between the overview we developed in the last chapter and this flowchart. In fact, that's the reason for not suggesting this approach to developing a flowchart. Remember, the idea here is to develop a program slowly. Much as wine needs to age, and different wines age at different rates, so too programs need time to develop. Some will develop quickly while others will take a while to be brought to fruition.

That's normal and desirable! Again, we're trying to develop a set of "program development guidelines" that will allow all programs to be produced in the minimum amount of time with the maximum yield. That means that we will force all programs to undergo a similiar, but not identical, maturation process. This process has to be flexible to allow for the individual differences and preferences. But to make one stage of development identical in content and different in form seems to defeat the purpose of everything we've learned so far!

Assuming that you are in agreeement with what we've said so far, let's continue to look at how a flowchart might be used. We've already mentioned that a flowchart should be part of the final program documentation, but does it have any other uses? Of course it does, or I wouldn't be mentioning it. Right? Of course!

The primary purpose of the flowchart is as a guide to the programmer as he is coding the program. The flowchart may be likened to a blueprint. It contains all of the program structural data, that is, it indicates the sequence of instructions, and the paths that should be taken based upon any logical or data related decisions. If properly developed, the flowchart should make the actual writing of the program a snap. The hard work should already be done. The other advantage of the flowchart is that it is an evolutionary step in the development of a program and, as such, it should be useful as a tool to help clarify your own thinking about the program.

As you develop the flowchart, it's easier to make changes at that stage than it is during the coding stage where changes may turn out to be massive. Again, the major reason is the difference in

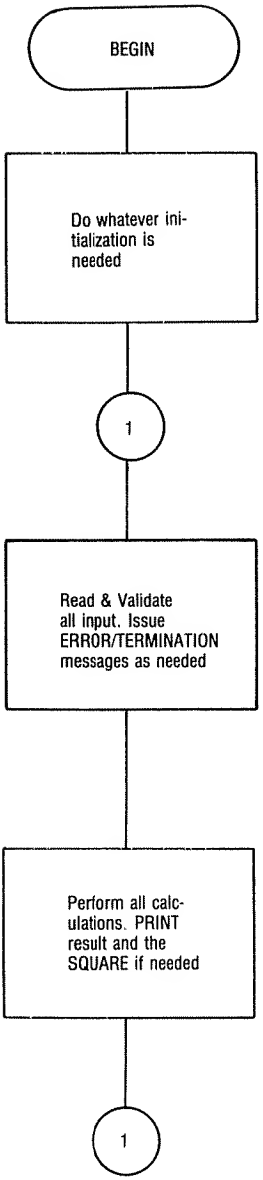


Figure 6-6. AREA Calculation Program Block Level Flowchart

amount of detail. As we get to each succeeding step in the development of a program we get more and more detailed. That is a normal step, and is the primary one that we are trying to cultivate throughout this book. Regardless of the steps that you will actually take, if you allow the program to grow slowly in a controlled environment you will find that the program will tend to be cleaner and work better sooner.

As you can see, the emphasis here is *not* so much on the flowchart itself, but on the tool! Just what do I mean by the "tool" if not the flowchart? Actually, flowcharts and all of that genre are designed for one reason only, as aids to proper program development. That's not to say that we can dispense with them, but rather that they serve as a means of focusing our attention on the problem at hand. This is much the same as an oriental mystic who will chant

a mantra during his meditation as an aid to more complete meditiation, attempting to bring his full concentration inward.

Concentration. Attention to detail. These are the primary tools that a programmer must have. But more than having a tool, the owner must also know how to use that tool. It serves no purpose to be able to focus complete attention on a problem in the early stages of development and then forget about it during the latter stages. That's another reason for the flowchart; it serves as a memory aid. As you go through the early stages of development you may be writing down various program functions. Once you start the flowchart you will incorporate these functions into a working picture of the program. During coding, all of the prior thought put into the program should be directly available to you in the form of the flowchart.

The flowchart is a pictographic representation of the program. Sort of like the models of the human body, you remember them, the "Visible Man" and the "Visible Woman." Well, the flowchart might be called the "Visible Program." Here in all its glory is the logic behind the program, laid out for all to see. That leads us to another reason for developing flowcharts. With the logic laid bare, there is a clarity of thought that takes place. We can very quickly see whether or not we have correctly evaluated the needs of the program. If there are connecting lines that should go someplace, but can't or if we don't know where they should go, we obviously have to spend more time thinking about the program. There's something missing in the design.

This can be considered sort of a "software logic probe." A hardware logic probe is used to test various components of a computer (integrated circuits and so on), and our software logic probe is used to test the program design. Actually, the analogy isn't all that hot because the logic probe for hardware is usually accurate while the software logic probe may not be.

Are there any other uses for a flowchart? Well, there's wallpaper, scratch paper, and so on. Seriously, there's always reasons for anything. We can consider just a couple of ideas that might make it more enticing to take the time to develop a good flowchart.

Let's start with one that's an outgrowth of an idea we've already presented. What do you do if, after several weeks of program development, you're suddenly unable to continue? Unreasonable? Not at all! If you're a student you might have final exams or, if you're a businessman, you might have to travel, attend a conference, or whatever. For some reason you can't continue with the development of your current program. Do you forget it? Not hardly! If you have a flowchart you don't really have to worry since the ideas are all set down in a form that is ready to translate to code. You can come back and pick up almost where you left off. Of course, there is some relearning time, but this should be minimized if you have followed all of the stages of development as outlined here.

A second reason is one that, at first, will sound quite trite, but I think you'll discover that there is some truth to it. Assume for the moment, that you are capable of never forgetting anything, regardless of how trivial. As you decide how the program should be written, you merely place a mental image of it in your "memory file." When you actually get around to writing the program, you recover this file and there it is, a program fully developed and ready to code! Sound

far-fetched? Not really, that's what a flowchart is. It will be there whenever you want it, and will allow you to translate the fully developed ideas into running code. The second reason, then, is that these flowcharts can become a file of programs to be written later on when there's more time available.

Now that we've covered *why* we want to develop a flowchart, are there any helpful ideas that will tell us *how* to go about developing a flowchart? Not really, let's talk about girls...

Of course there are! First of all, at the beginning of this chapter I mentioned that I was using a template. Just what the heck is a template? I'll bet you thought I wasn't going to define that one, huh! A template is, usually, a piece of plastic with the various shapes punched out. The one I use comes from IBM (whoever they are). It's available as part number GX20-8020. There is usually a "dash-number," or number following that part number, but it's not needed to order the part. If you want one, it can be acquired through your local IBM dealer (you know...the one you got your last System 370 through).

What about other tricks or techniques? Well, to be honest, there really aren't any books about flowcharting. So, any ideas given here will relate only to the way in which I use flowcharts or the way I've seen them used. The first step is to get a piece of paper big enough to write on (shopping bags don't work too well). Then get a pencil (not a crayon). By the way, a sharp pencil works better than one with no point! I try to set the flowchart so that the most logical functions appear first — either subroutines or the mainline, depending upon the school of thought I'm following.

The basic steps that follow are then to simply map out the logic blocks, don't worry about the connecting lines yet. Once you have filled up the piece of paper (which I find usually takes three columns of symbols) you can then go back and put in the connecting lines. Again, doodling fills paper but isn't what we're looking for! Actually, as you can see, its basically a mechanical task. To make it easier, let's take a look at the flowchart for our CHEKBOOK program. It follows most of the rules that I've espoused so far, and should serve as an admirable example (although I won't say whether it's a good or a bad example).

For simplicity sake, this flowchart follows a modular approach. We will look at how the program might have been built in a structured view in Chapter 9. In that chapter we will also present a possible high-level (minimimal detail) flowchart that will allow you to see the differences in thought behind the modular and the structured approaches to programming. I do, however, recommend a modified top-down approach to programming regardless of whether you use structured or modular techniques. We'll talk about the modifications to top-down in Chapter 7 where we'll be primarily concerned with pseudo-code, another development tool that may be used in lieu of flowcharts.

As you can see, there is very little in the way of dialogue associated with the flowchart for our checkbook program. That is as it should be. Since we know what the program is to do (from earlier chapters where we spelled out the requirements) we should not need much verbiage. That is, indeed, one of the primary purposes of the flowchart. So, read and discover. The program will follow but we'll first look at another way to develop a program without the use of flowcharts.

Why? Aren't flowcharts any good? Seems to be a strange question to ask at the end of a chapter on flowcharts, but I'd miss the boat if I didn't explain the good *and* the bad points. Flowcharts have one significant long-term drawback — they quickly get "out-of-synch" with the program unless they are maintained very carefully when the program is changed. That's one of the reasons I suggested using a slightly higher level of flowchart rather than a detail level that merely echos the code in the program. That will make the flowchart last longer since the changes will have to be more pronounced to have an effect on the program. Also changes will be relatively easy to make since only the major effect of the changes (normally) will have to be indicated. Obviously, if there are changes to decision blocks (which are always spelled out) this may require more drastic surgery to maintain.

But (here's the big rub) flowcharts *are* useful. They provide all of the nice things we've talked about so far, as well as being something that can fit right into the documentation package for the program. We'll talk more about documentation later, but this is really all we need to say now on it.

So, go ahead, use flowcharts as a development tool if you find it fits with what you are doing. As with all of the other material presented in this book, not everybody does it all the time. That old saying "but everybody's doing it" doesn't hold water here. So what? Even if *you* are the only one to use it, it could save you hours or even days down the road. You may not use it for every program, but, if you do use it when it's needed, it'll surely pay for the time it takes to develop it. After all, it's just a picture — maybe it can save the time it would take to write a thousand words.

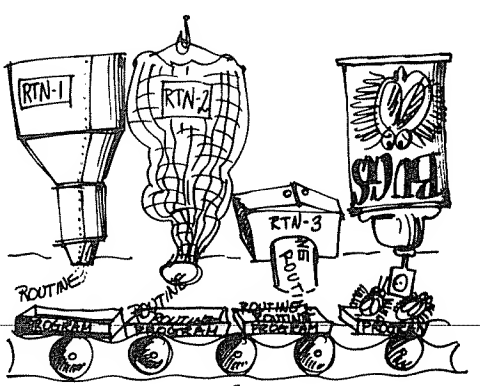
CHAPTER 7

Pseudo-Code

Now that we've gotten flowcharts fully mastered (we have gotten them mastered, haven't we?), let's press on with another way to assist in the development of a program. Flowcharts used pictorial symbols to define the logical flow of the program. Pseudo-code uses a different form of symbols — words — to accomplish the same purpose. Here we use words to describe the way in which a program must work. Sometimes these words may be the words that would have been contained in the symbols used on a flowchart and sometimes not. It doesn't really matter. We'll be looking at how we go about developing this pseudo-code so that it's meaningful to you and anyone that might be working on your program.

There are two different ways in which pseudo-code may be written. Our first one is sometimes called a "structural map" or chart of a program. Here we'll talk about hierarchy again, but this time at the module level. If we remember how we developed an outline of the program (way back in Chapter 4), we recall that we were looking at a hierarchical structure which we defined as a series of different levels or rows of importance. This first form of pseudo-code that we'll use is really a map that shows the way in which the various routines interact with one another, starting with the highest level and working down to the lowest functional levels of the program.

This approach allows the logical structuring of a program to be clearly seen. The purpose is, again, to make the programmer more



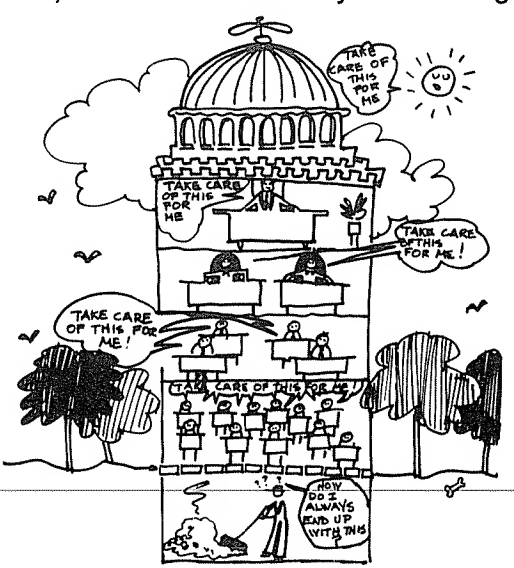

PSSSSSSSSSSSST...
IT MUST BE
PSEUDO-CODE!
DECRIPTION IN PROGRESS
ERROR!
ERROR!
ERROR!
GIGO

aware of the way in which the various routines of the program must interact. This module interaction is one of the primary points in a program where there can be errors. Since debugging is such a large part of the current development schedule of a program, it seems reasonable to spend as much time as possible "up-front" eliminating the bugs before they can occur.

That's one of the primary purposes for approaching program development in the fashion that we're describing in this book, to minimize the number of errors that creep into programs during the design and coding stage. Obviously, there will be some bugs since NOBODY is perfect. What we want to do, of course, is keep them to a minimum — the fewer bugs we put in the quicker we can get 'em out again (it says here).

Before we get too deeply involved in the mechanics of pseudo-code, let's take a quick look at the development of the tool so that we might be better able to see how it fits into the scheme of things. Pseudo-code is defined as a code that requires translation before execution. That's probably not terribly meaningful yet, but it will be!

Pseudo-code was developed as part of the so-called structured programming methodology. The entire thrust of this particular way of developing programs is that the program is thought out in a complete, logical fashion. We begin first at the highest level (or least amount of detail) and work successively down through the levels of

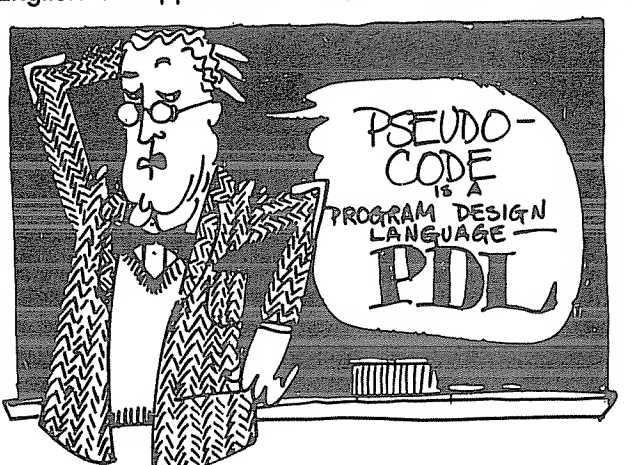


detail until we get to the greatest amount of detail at the bottom of the hierarchy chart. Think of it as a corporation; the top levels define what must be done, and pass these directions down to the various levels of underlings until the poor guys at the bottom (who do all the work) get the message. At each level only the amount of information needed to get the job done is passed to the routine that performs the actual task (sounds like the government).

The approach here was to make sure that any given routine was only as complex as that routine needed to be in order to perform it's particular function. Each routine would then be called a primative, or bottom element. For example, some primatives might be responsible for reading a record, writing a record, printing report headers (title lines and so on) on a fresh page of paper, or whatever. Each routine would only know what was necessary for it to perform its appointed task. For example, the routine to read a record would not need to know what page the report was on, and so that information would not be passed to it. Of course, in a BASIC program that data is available, but shouldn't be looked at — the variable containing it should never be referenced in the read record routine!

"Ok, but what, exactly, is pseudo-code?"

A fine question and one that we'll answer right now. Pseudo-code is a sort of programming language, but one that uses rather strange English as opposed to formal computer language. For



example, the following might be a way of expressing a routine that will repetitively read records until we find one that contains some specific condition (which is, apparently, undefined):

```
READ: routine
  DO WHILE (condition false)
    READ a record
  END (DO WHILE)
END READ.
```

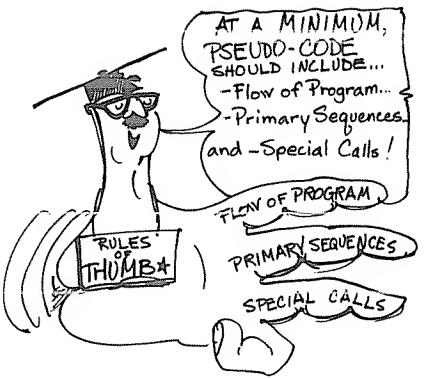
The routine is expressed in a fashion that seems quite simple, and yet the entire logic of the routine is clear to see. We don't need to know what the condition is at this level of code, that can be added as we develop the program still further. Remember, we are using this as a developmental tool, and it can be used throughout the development life cycle of a program.

Let's digress a moment to refresh our memory on the life cycle of a program. We begin with the idea which is then refined and, perhaps, merged with other ideas. This is then planned out and developed into a programmable idea. The requirements are defined and the program is designed. Following design the program is coded, tested, debugged, documented, and used!

The stages then are: need, definition, requirements, design, coding, testing/debugging, documentation, usage. There are some additional stages called, in general, "maintenance" which contain all of the initial development stages (only in miniature) and are used to enhance the program as the needs behind the original program change or evolve into something different.

Armed with this information let's see how what we know of pseudo-code could be used. Obviously we can use it in the design and coding stages as well as the maintenance stage. What about the documentation stage? Indirectly, yes! We can use the pseudo-code as part of the documentation just as we used the flowcharts of the previous chapter. Perhaps we can use a very simple form of pseudo-code in designing the testing cycles that we must go through. There doesn't really seem to be any other places that pseudo-code will fit properly. And that's also as it should be. We have a design tool, not a definition or requirements tool. Let's let it remain a design tool. Of course, design goes through many different levels, and so this tool, unlike flowcharts, would seem to be usable throughout the entire design stage.

Some guidelines are often given regarding how much material should be placed into a pseudo-code description of a program. As with flowcharts the amount of detail will vary depending upon the



individual, but there is what we can call a minimum set of data. This minimum set should include the basic flow of the program, the primary sequences from top to bottom, along with calls to the various non-linear routines (error, special condition, etc.). All logical choices made by the program should again be clearly spelled out so that when you begin coding the program there will not be any problem in writing the code once that phase has actually begun.

What we are discussing here is again a development methodology. This particular methodology is taken from development techniques like "top-down" and "bottom-up." There are as many advocates of this as there are for any other form of development. I'm not trying to imply that this is superior to any other technique. What I am saying is that these techniques have shown themselves to be efficient and useful in the development of programs. Other methods are also good and some are also efficient, but these seem to combine some of the best of all of the various development techniques that have been foisted upon the programmers of the world.

While we're talking about pseudo-code, let's also talk about another form of a chart that goes hand-in-hand with pseudo-code. Let's talk about hierarchy charts. We've already mentioned them, so let's cover exactly what they are and where they might be used.

A hierarchy chart is similar to a flowchart in that it uses pictures or symbols to represent various parts of the program. However, unlike a flowchart, the hierarchy chart uses only boxes. These boxes are aligned much like an organizational chart. In fact, they are organizational charts. These hierarchy charts reflect the logical organization of the program. They are used to graphically illustrate the various module dependencies. By that we mean that each module

is getting data from or giving data to another module. No module is independent in a program; it has some relationship with at least one other module in the program. This relationship is indicated via a connecting line between the boxes representing the modules.

Aside from the obvious usages of the hierarchy chart we can also use it as a checklist of modules to be written. Sort of a fail-safe method of guaranteeing that the routines are all present and accounted for (even though it won't guarantee that they will work). As a secondary check, it can make sure that you have coded the routines in the right sequence!

"How's that again?"

It will make sure that the routines are written in the order of their importance. That is, for top-down construction, you have coded the higher-level modules first and the lower-level modules last. This means that you can start testing while you are coding. You replace the lower-level routines with dummy routines that merely return; then you can begin testing. As you complete each logical layer you will be assured that any bugs that creep in will be either hidden in the linkage between the routines, or in the new routines that are yet to be added. Simple? It certainly is!

But let's get back to the pseudo-code. How does all of this relate to that? Let's take a good look at some simple pseudo-code for the AREA calculation program that we introduced back in Chapter 4. We'll start by drawing the hierarchy chart so that you'll be able to see the precise relationships between the various routines. We'll further assume that we're going to make this a "structured" program. I've placed the word structured in quotation marks because this isn't really structured, it's being done that way merely to facilitate this discussion. Here's our chart:

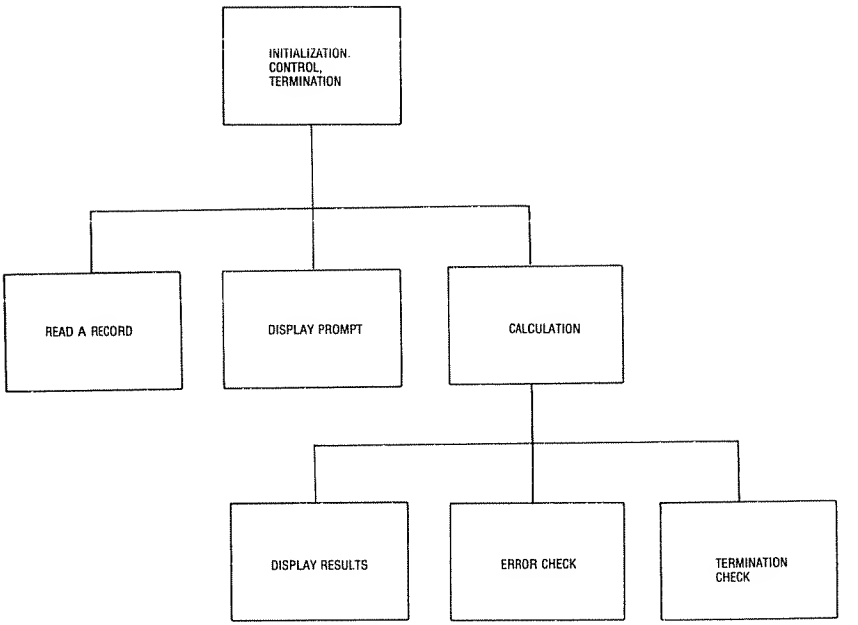


Figure 7-1. Hierarchy Chart for AREA Calculation Program

As you can see, there are three levels of hierarchy. The top level (called Initialization, Control, Termination) is responsible for setting up the correct environment (clearing the screen, initializing variables, etc.). It will call the "Display Prompt" routine which will request data from the user. After returning from that routine, our main level will call "Read A Record" which will get data from the keyboard. Data from the keyboard will be routed through "Control" to the "Calculation" routine. "Calculation" will call "Termination Check" (see if both width and length are zero) which will return control if we are at end. Assuming that this was not a termination condition, control will be passed to "Error Check" where a test will be made to see if either value was zero. An error message may be displayed if either value is zero; control is returned. If all went well the calculation is performed (in Calculation) and the results displayed via a call to "Display Results." Control is returned to "Control" for the next go around.

Obviously, all of that detail is not present in that small hierarchy chart, but I've added it so that we can see what is happening. Now let's take a look at a possible set of pseudo-code for this same program:

```
PROGRAM: Area
  Initialize all variables
  Clear the Screen
  While still processing:
    GOSUB Display Prompt
      PRINT "ENTER LENGTH AND WIDTH"
      RETURN
    GOSUB Read A Record
      READ L,W
      RETURN
    GOSUB Calculation
      GOSUB Termination Check
        Test for L=0 and W=0
        RETURN
      If end condition, RETURN
      GOSUB Error Check
        Test for L=0 or W=0
        If so, display error message
      RETURN
If error condition, RETURN
Calculate A = LxW
      GOSUB Display Results
    PRINT "AREA =" ;A
If L=W PRINT "THIS IS A SQUARE"
      RETURN
    RETURN
End While (Still Processing.)
Termination Processing.
```

As you can see, this is almost written in BASIC. It is clear that this was developed near the end of the development cycle since there is so much detail involved. Notice the indentations that allow the reader to immediately follow the logic. It makes clear what is happening in each routine. This is the same concept as the indentation we discussed earlier when talking about how a BASIC program should look from a purely physical standpoint.

Actually, this is more detailed than most pseudo-code should ever need to be. Also, it quickly becomes obvious that this trivial program should not need to be written in a manner that requires so much program overhead. By that we mean the structured approach has some program size limitations below which it costs more to use than it could ever return. One of the chief task that remains for the programmer is the need to be able to tell which program needs which development technique. This is why we said earlier that you could still retain your own personal artistic freedom in the development of programs while increasing your own productivity!

You may have noticed that we keep coming back to this idea of making a personal statement. Why? Because each programmer will bring a particular background, a particular understanding to the solution of a problem. It's that uniqueness that makes this field so interesting. We each have our own perception of the "correct" approach to writing a program to meet a particular need. No two of us will ever write a program in exactly the same way (with the possible exception of some very trivial programs).

Let's get back to pseudo-code and take a look at the ways in which it can be used. We've seen that we can make a fairly detailed description of a program using this technique, and we said that it can be used during the program design cycle. Let's see if we can follow a simple example through the design stages and see the way in which pseudo-code has helped us develop the program. For the sake of simplicity we'll be using our standard example of the AREA calculation program. Our first iteration (iteration means to make a pass through some cyclic process, an iterative process is one that requires many passes to complete) will be at a very high level. It might look like this:

```
PROGRAM: Area
  Initialize
  While still processing:
    Display Prompt
    Read A Record
    Calculation
    Display Results
  End While (Still Processing.)
  Termination Processing.
```

As you can see, this retains the structural requirements of the program while not containing a large amount of detail. This is a reasonably good first pass at designing the program. While we could write the program from this data we might miss something. There's no mention of the termination condition or of any special checks for errors or a square. While not terribly significant for this small, almost trivial, program it could be important for a larger, more complex program.

Let's continue with the evolution of this program and see if we can't determine the next stage or iteration. Adding the checks we've just mentioned ought to add just the right amount of detail. It might the look like:

```
PROGRAM: Area
  Initialize
  While still processing:
    Display Prompt
    Read A Record
    Calculation
      Termination Check
        Test for L=0 and W=0
        RETURN
      If end condition, RETURN
      Error Check
        Test for L=0 or W=0
        If so, display error message
        RETURN
      If error condition, RETURN
      Calculate A=LxW
      Display Results
        If L=W PRINT "THIS IS A SQUARE"
      RETURN
      RETURN
  End While (Still Processing.)
  Termination Processing.
```

This has now become fairly detailed, merely by adding the special checks for the termination condition and for a square. Of course, if another iteration is required we would then arrive at the example given when we first introduced this concept.

Now that we've done this for a small program, let's do it for our major program. We've already laid out the requirements for the checkbook program so we can quickly build a minimal level of detail in pseudo-code. We can also build a hierarchy chart which may help us to determine just exactly how we want to structure our program. While we're talking about structure, let's bring out another very important point. We've mentioned the two schools regarding placement of subroutines while not taking a stand on either. And yet there seems to be an implicit approval given to the idea of placing subroutines last since that's where they wind up in a hierarchy chart. Actually, for most BASIC interpreters it makes sense from a performance standpoint to place the subroutines first (since most of them start scanning from the beginning of the program to find a line number reference). We will allow you to decide where you want to place them, and please don't assume that any recommendation is implied by the use of hierarchy charts or any other tool. Now, let's get on with our pseudo-code for the checkbook program. Our first pass might look something like this:

```
PROGRAM: Checkbook
  Initialization
  While processing data
    Prompt for data
    Validate data
    Process
      Post a Check
      Write a Check
      Print Account Statements
    End Process
  End (while processing data)
  Termination routines
```

This rather simple set of statements contains most of the necessary routines for the program, at least there is a statement for each. Addtionally, there is a lot of room for growth! As we go through the iterative process of evolving our program design, we'll be adding more and more detail to this.

Once we've got the basic logic down, we can then begin by expanding to the next level of the hierarchy. At the level below the

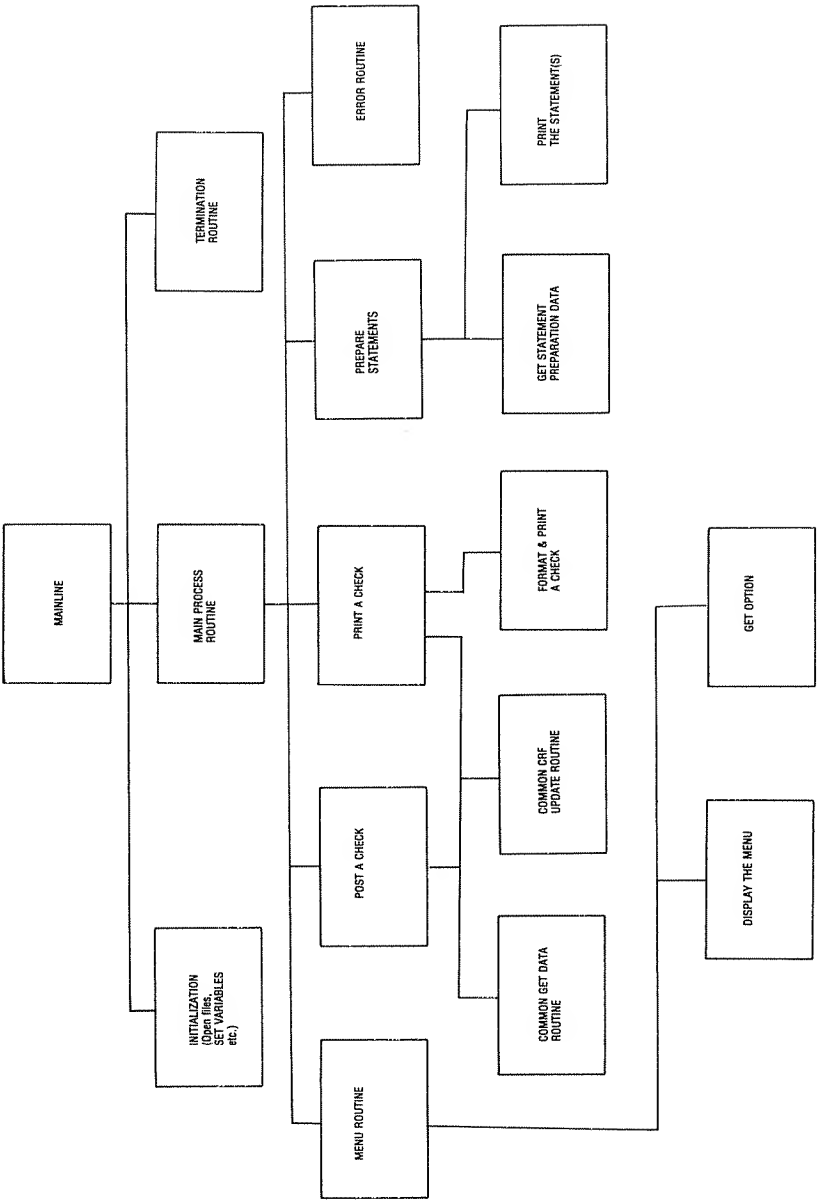


Figure 7-2. CHECKBOOK Program Hierarchy Chart

major routines, we will have a more detailed statement about the first level plus some information about the second level and a single line about the third level. This increasing amount of detail continues until we arrive at a final product. The final product should allow us to completely write the program without spending any time deciding how the program should be constructed. In fact, once we've reached this stage the rest of the program development (the coding) could almost become a purely mechanical task!

If we go on to the next level of detail, we might wind up with pseudo-code on the order of:

```
PROGRAM: Checkbook
  Initialization
    Initialize variables
    Open all files
  While processing data
    Prompt for data
   Validate data
      Is this a valid option?
    Process (based on option)
      Get required data
      Validate data (based upon option)
      Post a Check
      Write a Check
      Print Account Statements
    End Process
  End (while processing data)
  Termination routines
    Display any final messages
    Close files
```

As you can see, we've added logic for selecting routines to be used in the main processing area, as well as getting data for both option selected and the data needed for each of these routines. We won't go much further in developing this now. Instead, we'll take a look at the hierarchy chart that might be associated with this program at this stage.

It's easy to see that there are four levels of hierarchy reflected in the chart. Looking back at the pseudo-code and counting the levels of indentation we find (Surprise! Surprise!) four levels of

indentation. How about that! There really is a correlation between the pseudo-code and the hierarchy chart that is readily apparent. While we are looking at this revelation (which you probably guessed would hold true) we ought to make one other observation. The format of the hierarchy chart and the pseudo-code both reflect an arbitrary first level, one called PROGRAM which corresponds to MAINLINE in the chart. This is merely a convention that allows lower-level structure. It isn't really a level by itself, but may be considered a box into which the program can be placed.

By the way, you may recall that we mentioned a charting technique called HIPO in the last chapter. Well, the hierarchy chart that we've been using is actually based upon that charting technique. Remember, HIPO stands for Hierarchy plus Input, Process, Output. We haven't really looked at the IPO part of HIPO nor shall we. The reason is that it isn't essential to the discussion. That's more of a documentation tool than it is a design tool even though it has been used for design, and is promoted as a design aid.

One idea that needs to be stressed, and perhaps should be written on every computer terminal and CRT is that programs must be properly designed before they can be properly coded. That seems like an obvious statement, and yet time and again it has been overlooked. Programs have been developed without any real or formal attempt at designing it first. Now that we've examined some of the design techniques and seen that they can be useful (if somewhat time consuming), we can understand that there is a trade-off. Beyond a certain point excess time spent in design is not going to be profitable. On the other hand, not spending enough time in design may result in excess time being spent in the coding and debugging stages of the program development cycle.

This design cycle is real! It's not an imaginary thing made up by writers of books and magazine articles designed to confuse unwary programmers (or their management). If we look at the way in which program development has grown from an infant "science" to the present day state, we can see some good and a lot of bad that has gotten us to where we are. If any other function had produced as many unhappy users, as much rubbish, and as many errors as has the programming function, it would have long since been discarded. As it is, we survive as an entity because there is: (1) a need for us and (2) we have our own machines!

However, neither of these two reasons are valid excuses. We need to be aware that our skills can be sharpened by using a more scientific approach to program development, and that's the purpose of the material we've covered so far. As we continue in the book, we will be looking at ways to apply the design ideas that we've picked up. We'll be taking the design critera for our checkbook program and actually coding the program. We'll code it using both the "flow-through" or straight line method of coding as well as the structured "top-down" approach. This contrast should allow you to get a better feel for the two approaches, and decide with which you feel most comfortable.

We've talked about the possible impacts of poor design on the end product. Let's stop for a moment and explain what happens to the design/coding cycle as a result of poor design. We can begin by assuming that we have a rough idea of the program content and its functions; so we begin coding. As we encounter difficulties in coding a particular routine we may have to stop and rewrite some code to make it fit better. Or perhaps we may have to stop to duplicate some code changing some variable names because we didn't think that a given routine might be needed more than once. That's twice the number of routines to be debugged for each routine that is duplicated; twice the number of chances for errors to creep in.

Once the program is coded, we start testing and find that it doesn't quite work the way we want it to, so we go back and make a few changes. They work, but now some bugs appeared in other areas of the program. Strange, those changes *couldn't* have caused that problem. Back to the coding stage, let's get that bug! More coding and that bug goes away but now there's a new bug. Gotta rewrite a routine to eliminate that bug, but that affects the routine interfacing logic so there's still more changes.

I think you get the point. As a wise man once said, if a thing's worth doing, it's worth doing well. From our standpoint that could be interpreted as saying that we ought to only code the program once; *after* we've designed the program, not before.

Let's make a few final points on program design as we close out this section on design. We've presented three techniques that you can use — flowcharts, pseudo-code and hierarchy charts. Of these techniques, flowcharting is the oldest and least effective way of designing programs in the "modern" style. By that we mean that flowcharts will reflect the flow-through style of programming by showing the flow of instructions while not indicating any program structure other than, possibly, modules.

There is nothing wrong with this if you plan on utilizing the modular coding techniques. If, however, you are planning on using the newer structured techniques you might want to consider using the hierarchy chart and pseudo-code. These two techniques work well together and allow a rapid and (almost) painless way of designing a program. One advantage here is that the structure of the program is made obvious from the design tools used, either through the indentation of the pseudo-code or through the levels which are so visable in the hierarchy chart.

Unfortunately, none of these techniques will guarantee you a good working program. They are nothing more than tools of the trade. For a writer one of the tools that is available is the "typewriter" which is used to produce an end product, words. This tool has a learning curve associated with it because of the need to be able to touch type. Obviously, the tool may be used by someone that doesn't know how to touch type (I don't) but it may be less efficient. I learned my own style of typing and can type better than 65 words per minute

once I get warmed up, but I make more mistakes while I'm getting warmed up. The key here is that we must learn to use the tools we have. We may not use them the way they're intended to be used (such as using pliers as a hammer) but we can get the job done. The better we use the tools the better the job (pliers may bend the nail).

These techniques were formulated as ways to allow the programmer to maximize his efforts, that is, to get the most out of the effort put into developing a program. I mentioned at the beginning of this book that these were techniques that the really good programmers used all the time, and we were simply getting the benefit of people that had observed the good programmers and written down what it was that they did. What I omitted (and they did too) is that really good programmers don't just jump in. They take their time and make sure they know where they're going. They get the "road maps" that allow them to know the route they are to take. They know what the user expects the final product to do, and so they work toward that as a goal. They don't mistake getting a program up and running as a goal, it isn't. The goal must always be to provide what the user wants.

We discussed the user before, but each of us is a user. We write programs that we want to use. If we are doing this so we can market the end result, we do a market study to determine what the market wants, and then the "market" becomes the user. In all cases we need to produce a product that will make the user happy. If we're the user, we don't want to spend all of our time rewriting the program to do what we want (we won't be happy doing that). If the end user is a customer, we want a user that's so happy that others will come and purchase our product. We all like money, right? By now we've all gotten the message. We design the program, we code the program, and we enjoy the fruits of our labors. With any luck, that's exactly the way it will work. All we have to do is follow the guidelines set out for us by the experts in the field. Not because they're experts (which, by the way, comes from the Latin "ex" meaning "out of" and "spurt" meaning "a drip under pressure"), but because they're right!

Section 4 — Coding the Program

We've now reached the point in our program development cycle where we will actually start coding the program. In this section we will take a look at two of the more popular coding schemes. In Chapter 8 we'll be looking at modular coding techniques. This will allow us to immediately remove ourselves from some of the very early coding styles which were less conducive to effective program development.

In Chapter 9 we'll be looking at ways to take larger programs and use the most popular of the current program coding schemes (structured programming). We'll remain somewhat neutral in our observations regarding both of these schemes as well as the current great argument regarding the presence or absence of the verb "GOTO."

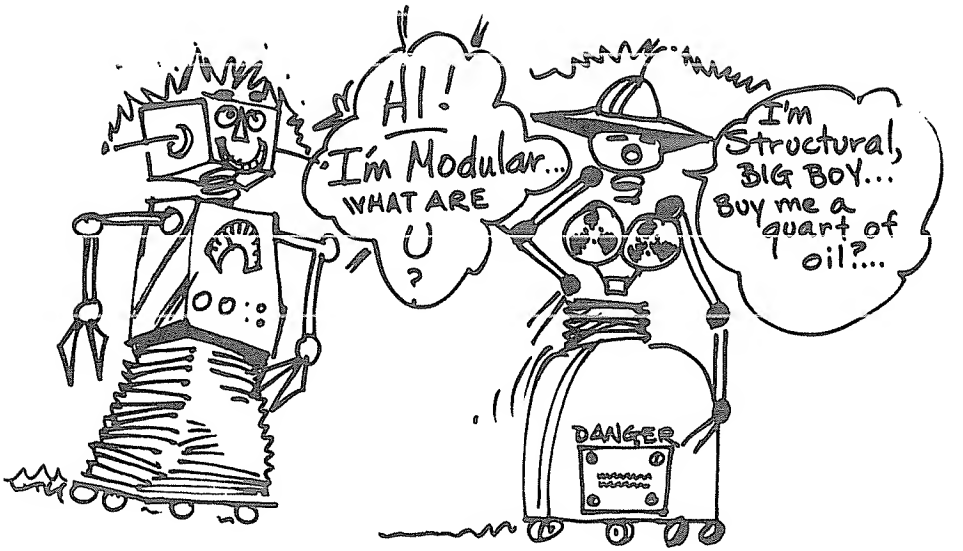
Throughout this section we'll actually be writing our checkbook program. To make the comparison of these two development techniques more objective, we'll actually write the same program utilizing both techniques. That should allow us to be able to decide which technique is more appropriate for our own particular needs and desires.

CHAPTER 8

Modular Coding Techniques

The term modular calls to mind all kinds of nicely packaged ideas. In fact, that's exactly what we're looking for — nice packages! We've used the term modular to refer to prefabricated houses (modular housing), to truck bodies placed on trains for long distance shipping (modular shipment units), and so on. In each case we're looking at the idea of a closed, self-contained unit.

When we speak of modular coding we're also talking about a self-contained unit. This unit is, however, made up of lines of program text or code. In Chapter 4 we introduced the idea of a function saying that a function was the same as a routine and a routine was the implementation of an idea. What that boils down to in the current context is that a module is a routine. This is a nice, convienient quantum or unit of code.



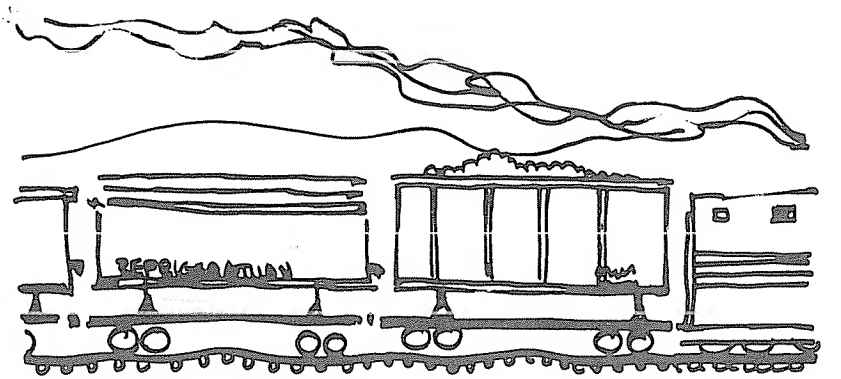
Now, if we can define units of code that can be structured as modules, what more is there? Actually, being able to break the code into modules is only a small part of the task. We must also be able to interface these modules correctly. Let's take a simple example, a modular stereo system. If we have a tuner, turntable, amplifier, and tape deck, we have four units which must be interfaced. The tuner, turntable, and tape deck need to be interfaced with the amplifier. The amplifier, in turn, needs to be interfaced with speakers.

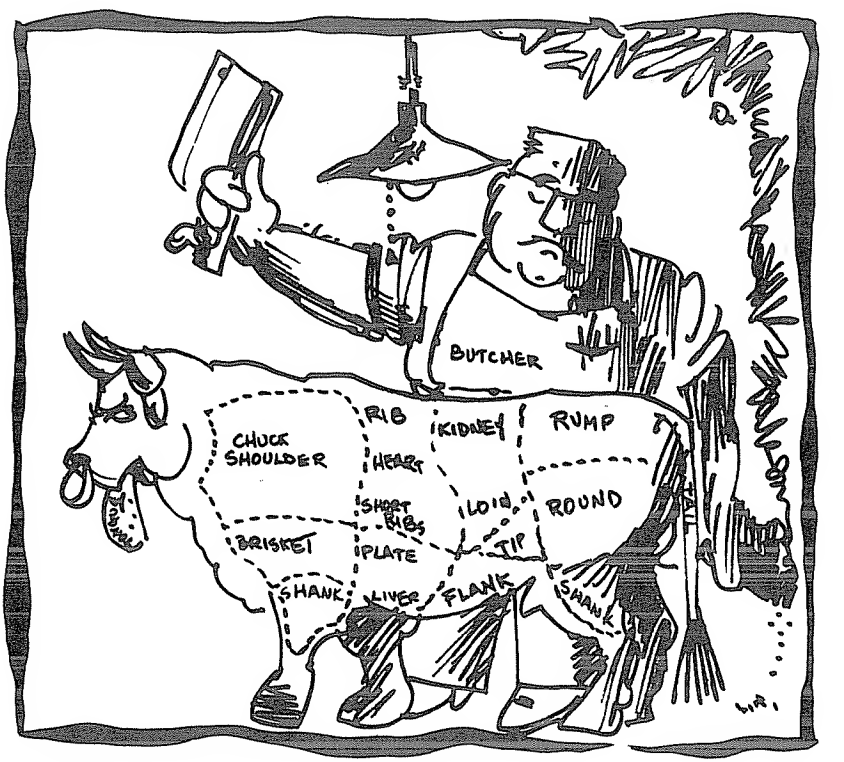
The interfacing requires that we have the correct kind of plugs on the end of the cables. If we have DIN plugs on everything but the amplifier, and the amplifier expects phono plugs we have a problem — it ain't gonna fit! If our amplifer produces 250 watts of output power we know that it was designed to be fed into low-efficiency speakers (or else it would blow the windows out of our house). If we only have high-efficiency speakers we have another kind of problem — we'll blow the speakers apart if we run the volume too high!

Interfacing is critical! We must make sure that we can properly communicate with each module. If we have a Frenchman and a German speaking to each other in their own native languages, we have a cacophony (a raucous noise), but not a dialogue (unless they both understand the other's language). Again we see that correctly communicating is a function of proper interfacing.

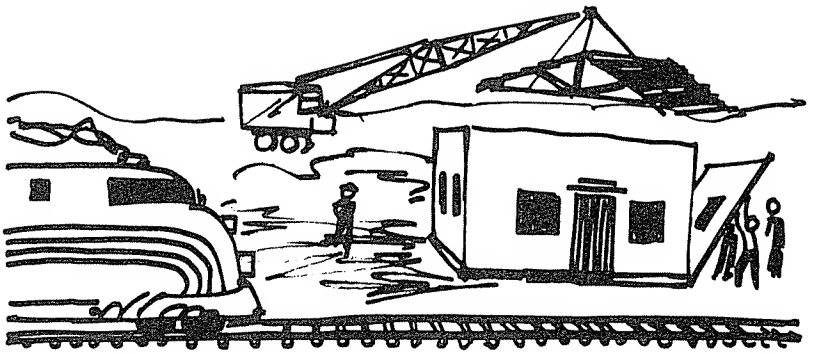
The idea of modular code is really great, but there are all kinds of requirements that must be met. Modular coding was designed to allow large programs to be written by many different programmers, each responsible for a certain number of modules. This required that each module have a defined interface, a list of the data that was required to correctly use the module. When this information was either incorrect or, more commonly, ignored by the programmers, there occurred a large number of programming errors. The program didn't work as designed.

In spite of the fact that we're now at the coding section of the program, we're still talking about the design of the program. This discussion is still applicable here since design can, and should, continue during the coding cycle. Specifically the design that has taken place so far has always been at an increasingly greater level of detail until the time when we have enough to begin coding. At the start of the coding cycle we should have enough information to decide on the modules that will be in the program as well as the interfaces to these modules. This is the kind of design we're now talking about, and it is properly considered as part of the coding cycle.





Before we actually get into writing our checkbook program let's talk about some of the principles of modular coding. We'll be discussing the way one actually implements a program using modular techniques so that it will make the examples that we'll be using later



on clearer. If we decided to use either the hierarchy chart or the pseudo-code, we will already have the basic modular structure that is needed. With flowcharts, we will have some very large modules. We'll take a look at how we'd use flowcharts first.

At each dividing point in a flowchart we will wind up with a CONNECTOR (the circle) that will set off certain parts of the flowchart from other parts. We can use these as the delimiters (things that set off or mark the boundries) of a module. In some cases this will result in fairly large modules, but if the entire block of code seems to be related by performing the same function, that's all right. If there seems to be more than one function included in this block of code, then try breaking it up by logical function. Actually, we're looking for logical groupings which can be considered as functions or routines. If we can successfully break the flowchart up into these separate routines we'll be well on our way to writing a modular program.

This process should be fairly easy for the special condition and error routines. They're already pretty well broken up by virtue of the fact that we jump out of the main line of code to get to them. Of course, there is still a CONNECTOR symbol at the top. By the way, the bottom of a routine may be indicated either by the presence of another CONNECTOR or by a TERMINATOR. Normally either of these would indicate a break in the logical structure. Since there is an exception to every rule, let's get it out of the way as quickly as we can. Sometimes we may draw a flowchart that has some code which we jump around because of some data dependent conditions. This may require the use of a CONNECTOR, but should not indicate the end of module. A better way to represent this condition is to draw

the code that will be executed as a result of the compare (or whatever it is) directly in line and connect the other leg of the compare (or whatever) directly to where it re-enters to main stream of the code. This will eliminate an unnecessary CONNECTOR as well as keeping the flowchart neater and easier to read. It will also mean that the flowchart is a better representation of the program!

If you chose to use the hierarchy chart the logical modular breaks are automatically shown. They will correspond to the various blocks on the chart — very straight forward and easy to visualize! With pseudo-code there is also an explicit grouping of code that will form a given module. This grouping will require some care since there is always a tendency to take more than should be placed in a module. The basic rule here is that each level of indentation forms a module. The exceptions to this are, unfortunately, numerous. First, let's make sure we understand the intial statement. We take any given level of indentation and that forms the module. We do not take the material at a lower level just because it is between two areas with the same indentation. For example, look at the rather simple pseudo-code structure below, it contains only letters. These letters represent the module into which the code will be placed.

```
A
   B
   B
      C
      C
   B
```

As can be seen, there are three statements that will go into module "B," two that go into module "C," and one dummy module called "A."

This sort of structure can also lead to grabbing too much material at any given level. For example, the "B" level in the chart above is going to be quite popular for a lot of material, but not *all* of that material belongs in one module. We have to be very selective in determining if it all fits into the same logical routine. If it doesn't then it should be placed into a different module (but at the same level of hierarchy). This is where the flowchart and the hierarchy chart have a clear-cut advantage over the pseudo-code.

Of course, any discussion of this nature is only academic. Since the design process will probably bring out the identities of the mod-

ules quite clearly, there should not be any difficulty in building modules appropriately! Right?

We've discussed a "dummy module" a number of times. Just what the heck is this dummy module, and how do you go about coding it? To make life simple we'll consider the dummy module to be the variable definitions — the DIM and other related statements. This will not hold true for languages other than BASIC but it'll work just fine for us for right now! Actually, as we mentioned earlier, the dummy module really doesn't exist, it's just there so that we can have an arbitrary "level 0" or beginning for our pseudo-code or hierarchy chart. We don't need it for a flowchart.

Now that we've all absorbed this theoretical material, let's get on with the nitty-gritty. We'll actually begin coding the program!

"Well, where do we begin? This book was supposed to tell us how to write a program and we don't even know where to begin!"

Hey! Foul! We've already begun. We began when we did the program design. Now we're merely going to take that design and implement it in a programming language. The first thing that we should do is to establish a set of variables. Each set has a different function. For example, we need loop control variables, data variables, special purpose variables, and so on. For the sake of my programming roots (which are located in FORTRAN) I always use the variables "I" through "N" for loop control. Variables "Ix" throught "Nx" are not reserved for this purpose, and may be used in other ways.

Again, the reason for reserving these variables and defining them up-front is to eliminate any guessing that might take place later on during the coding stage — have I used that variable already? We want to plan the code as carefully as we planned the program, and this is another reflection of that planning.

Let's restate the functional blocks of code that we've decided we will be needing. First we have the INITIALIZATION routines (establish dimensions, open files, load inital data, and so on). We'll need a MENU routine that displays the various options available to the user and validates his response. The actual processing routines that are needed are the POST routine to post manually written checks, the WRITE routine used for computer generated checks, the STATEMENT routine which will produce the check register statements.

Now, since we've carefully designed the program, we know that these are the only routines needed, right? Wrong! As usual, the

writer (me) has kept some secrets which we'll now consider. We need a way to set up the accounts in the first place, or as new accounts are added to the program. Remember, the program can process up to five accounts in our initial implementation. So, we need a routine called SETUP. Any more? Yup! We need a way to make a DEPOSIT to the account, and a way to PURGE the data from the Check Register File. The END routine should also make sure that the CRF is updated as a result of any changes made during the processing cycle.

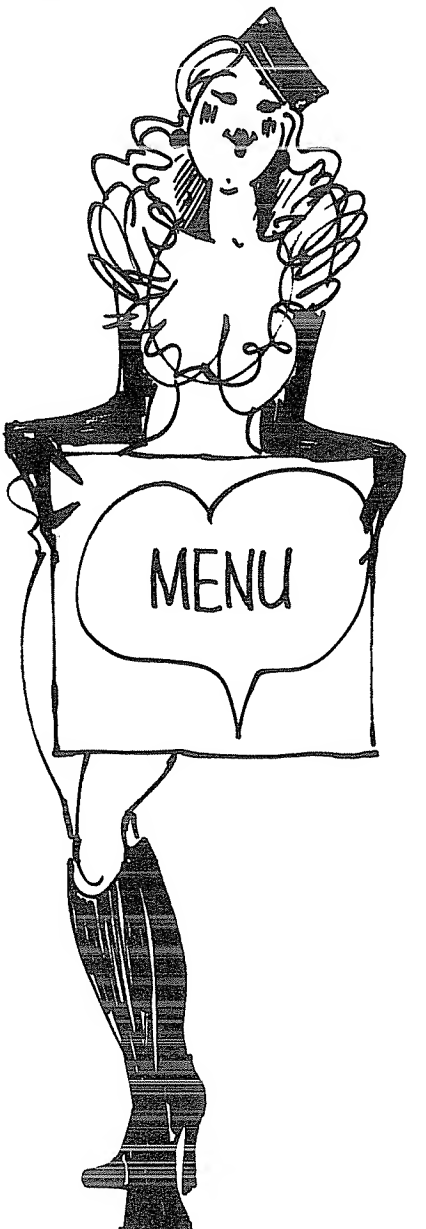
Now are there any more? No, that's everything (I think). Actually, we'll see if we've thought of everything as we go along. Design oversights usually come up during testing, that's when the user gets to see what's being made available so that he can verify whether or not it will meet his requirements (and his sometimes unstated needs).

Let's begin the coding, then, with the INITIALIZATION routines. Here's what might be needed:

```
10 REM              CHECKBOOK PROGRAM
20 REM  This program will process multiple accounts (w
   ith the limit currently set to 5).
30 REM
40 CLEAR 4000                        'RESERVE STRING SPACE
50 DEFDBL A
60 MN=150                            'CHECKS PER MONTH
70 MA=5                              'MAX ACCOUNTS
80 DIM CR$(MN,MA),AB(MA),AN(MA)      'Chk Reg, Acct, Bal,,
   Acct #
90 DIM CN(MA),NU$(25),MO$(12)        'CHK NOS, AMOUNTS, MO
   NTHS
100 REM ------------------------------------------------
110 REM         OPEN FILES, PROCESS DATE
120 REM
130 CLS
140 T$="CHEKBOOK -- A CHECK ACCOUNTING PROGRAM"
150 PRINT TAB(32-(LEN(T$)/2))T$
160 PRINT : PRINT "SYSTEM INITIALIZATION IN PROCESS, P
    LEASE WAIT"
170 ON ERROR GOTO 250
180 OPEN "I",1,"CRF/DAT" : ON ERROR GOTO 0
190 FOR I=1 TO MA : INPUT #1,AN(I) : NEXT I
200 FOR I=1 TO MA : INPUT #1,AB(I) : NEXT I
210 FOR I=1 TO MA : INPUT #1,CN(I) : NEXT I
220 IF EOF(1) CLOSE : GOTO 300
230 INPUT #1,I,J : LINE INPUT #1,CR$(I,J)
240 GOTO 220
250 PRINT "ERROR, CHECK REGISTER FILE NOT FOUND, FILE
    WILL BE CREATED!"
260 PRINT "HIT ENTER TO CONTINUE"
270 A$=INKEY$ : IF A$="" GOTO 270 ELSE IF ASC(A$)<>13
    GOTO 270
280 RESUME 290
290 ON ERROR GOTO 0
300 DT$=LEFT$(TIME$,8) : GOSUB 2510
310 IF VAL(DT$)<>0 GOTO 340
320 LINE INPUT "ENTER TODAY'S DATE (MM/DD/YY): ";DT$
330 GOTO 310
```

As we can see, lines 10 through 330 take care of most of the initialization logic. We do have a GOSUB 2510 which loads some special values used in printing checks which we'll talk about in a moment. So far, everything is fairly straight-forward. We should note that this program will work for both disk and cassette based systems since the CRF (Check Register File) is not updated directly. The data is read into the various arrays during initialization and will be rewritten at the end of the program as part of the termination routine. If you are using a cassette based system, replace the "INPUT #1" statements with "INPUT #-1."

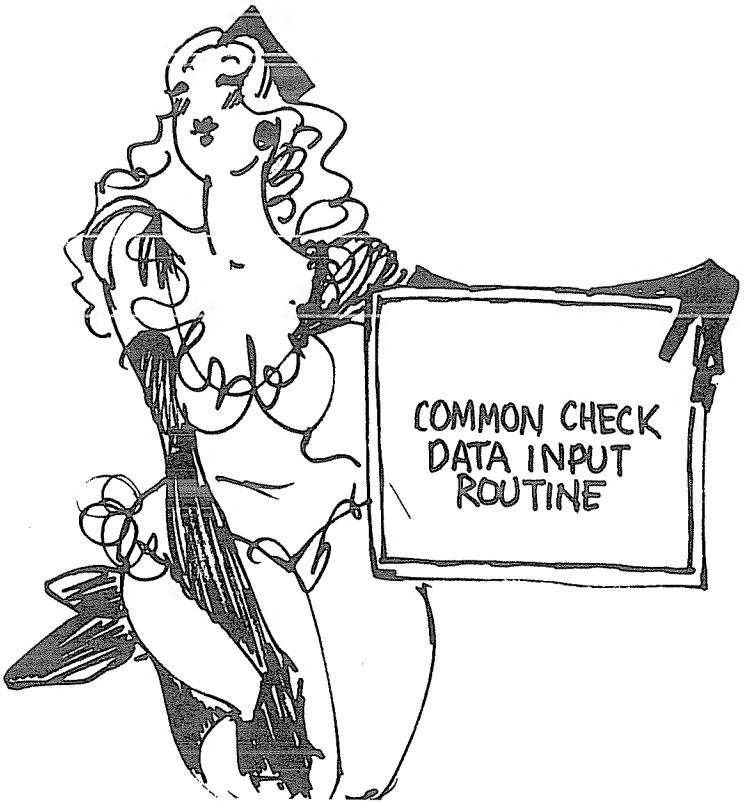
The MENU routine is another very simple piece of code that displays a menu similar to that described in Chapter 5. All of the main options (program functions) should be displayed so that the user can select the main option he wants. If there are other, lower options, they should be selected from a different menu screen. Here's this routine:



```
340 REM ------------------------------------------------
350 REM  DISPLAY MENU, GET & VALIDATE OPTIONS
360 REM
370 CLS
380 PRINT STRING$(24,"-")" CHEKBOOK MENU "STRING$(25,"
    -")
390 PRINT
400 PRINT "      1 -- POST        POST A HAND WRITTEN CH
    ECK"
410 PRINT "      2 -- WRITE       WRITE A CHECK"
420 PRINT "      3 -- STATEMENT   GENERATE THE CHECK STA
    TEMENTS"
430 PRINT "      4 -- SETUP       SETUP A NEW ACCOUNT"
440 PRINT "      5 -- DEPOSIT     POST A DEPOSIT TO ACCO
    UNT"
450 PRINT "      6 -- PURGE       PURGE CURRENT MONTH'S
    CRF"
460 PRINT "      7 -- END         TERMINATE THE PROGRAM"
470 PRINT@ 182,LEFT$(TIME$,8);
480 PRINT@ 246,RIGHT$(TIME$,8);
490 PRINT@ 64,"SELECT OPTION ===> ";CHR$(14);
500 A$=INKEY$ : IF A$="" GOTO 500
510 PRINT A$;CHR$(15); : OP=VAL(A$)
520 IF OP>=1 AND OP<=7 GOTO 550
530 PRINT@ 50,"INVALID OPTION";
540 GOTO 490
550 ON OP GOSUB 940, 1240, 1300, 1860, 2070, 2810, 227
    0
560 GOTO 370
```

Lines 340 through 560 form the menu routine. Note that the variable used to read the option (OP) is more or less meaningful (within the constraints of BASIC). This convention is used to make it easier to remember the variables that are being used throughout the program. We'll see why this is important later.

Our next block of code that we'll be writing, just for the heck of it, is one we haven't discussed directly. If we stop and think about it, just for a moment, we'll quickly realize that it makes sense. The routine we're talking about is a common routine used to get the information for writing checks. That makes sense. We have two different routines (POST and WRITE) that need basically the same information so rather than write two separate routines we can write one common routine that meets the needs of both. We can test some variable to see which routine invoked us, and that can be used to make the decision on the various pieces of data that we'll need. Since it takes longer to talk about it than to write it, here it is:

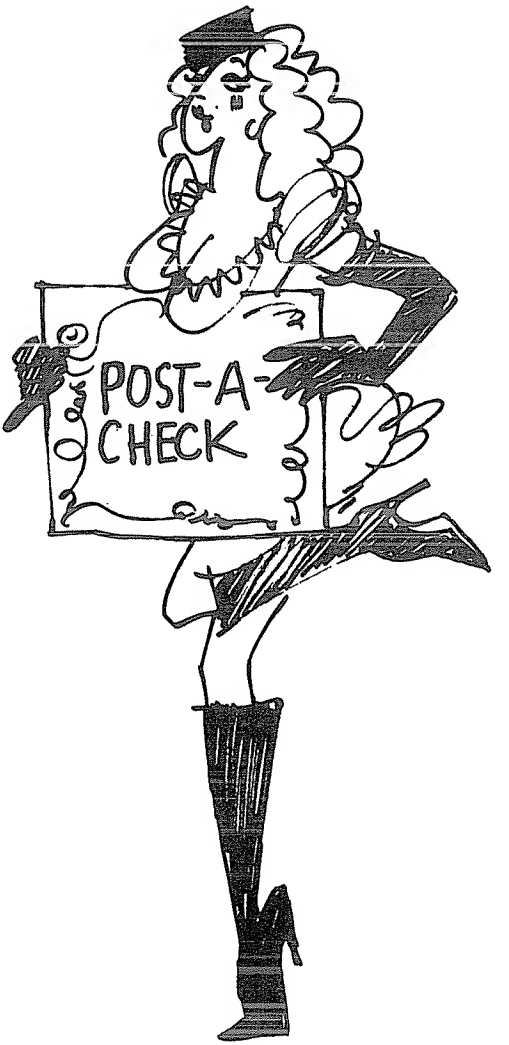



```
570 REM ------------------------------------------------
580 REM     COMMON CHECK DATA INPUT ROUTINE
590 REM
600 PRINT@ 128,CHR$(31)
610 PRINT@ 128,"";
620 CD$(1)="      ACCOUNT: "+STRING$(15,CHR$(95))
630 CD$(2)="      WRITTEN TO: "+STRING$(25,CHR$(95))
640 CD$(3)="      CHECK AMOUNT: $"+STRING$(7,CHR$(95))
650 CD$(4)="      NOTES: "+STRING$(25,CHR$(95))
660 DX$=STRING$(2,CHR$(95)) : DX$=DX$+"/"+DX$+"/"+DX$
670 CD$(5)="      DATE WRITTEN: "+DX$
680 CD$(6)="      CHECK NUMBER: "+STRING$(7,CHR$(95))
690 CN=-1
700 FOR I=1 TO 4 : PRINT CD$(I) : NEXT I
710 IF CC=1 PRINT CD$(5) : PRINT CD$(6)
720 PRINT@ 142,CHR$(14); : GOSUB 840 : AC=VAL(IX$)
730 PRINT@ 209,CHR$(14); : GOSUB 840 : WT$=IX$
740 PRINT@ 276,CHR$(14); : GOSUB 840 : AM=VAL(IX$)
750 PRINT@ 332,CHR$(14); : GOSUB 840 : NT$=IX$
760 IF CC<>1 GOTO 800
```

```
770 PRINT@ 403,CHR$(14); : GOSUB 840  : DW$=IX$
780 PRINT@ 467,CHR$(14); : GOSUB 840  : CN=VAL(IX$)
800 PRINT@ 512,""; : INPUT "IS THIS CORRECT";AN$
810 IF AN$="NO" OR AN$="N" GOTO 600
820 IF AN$<>"YES" AND AN$<>"Y" GOTO 800
830 PRINT@ 512,CHR$(31); : RETURN
840 IX$="" : A$=INKEY$
850 A$=INKEY$ : IF A$="" GOTO 850 ELSE IF ASC(A$)<>13
    PRINT A$;
860 IF ASC(A$)>31 THEN IX$=IX$+A$ : GOTO 850
870 IF ASC(A$)=8 THEN IF LEN(IX$)>0 THEN IX$=LEFT$(IX$
    ,LEN(IX$)-1) : GOTO 850
880 IF ASC(A$)=24 PRINT " "; : FOR I=1 TO LEN(IX$) : P
    RINT CHR$(8); : NEXT I : IX$="" : GOTO 850
890 IF ASC(A$)=13 PRINT CHR$(15); : RETURN
900 GOTO 850
```

The code between lines 570 and 900 takes care of getting the common information; the account number (for posting purposes), to whom the check was written (also called the PAYEE), the amount of the check, any notes or comments about the check. Some optional data are the date written and the check number (used by the POST routine).

Since we've just developed the routine to handle the common check input routine, let's go ahead and write the POST and WRITE routines. We'll write the POST routine first since we can then use some of the code in the POST routine for the WRITE routine. Specifically, we can use all of the routines that find the entries for the appropriate account, create the entries for the check register file as well as the routine that adjusts the balance of the account. So, without further ado, here's the POST routine:




```
910 REM ------------------------------------------------
920 REM      POST A HAND WRITTEN CHECK HERE
930 REM
940 IF AN(1)=0 GOSUB 1860
950 CLS
960 PRINT STRING$(20,"-")" CHECK POSTING ROUTINE " STR
    ING$(21,"-")
970 CC=1 : GOSUB 600
980 FOR I=1 TO MA : IF AC=AN(I) GOTO 1020 ELSE NEXT I
990 PRINT "ERROR, ACCOUNT NOT FOUND. RE-ENTER."
1000 INPUT "ACCOUNT NUMBER";AC
1010 GOTO 980
```

```
1020 FOR J=1 TO MN : IF CR$(J,I)="" GOTO 1050 ELSE NEX
     T J
1030 PRINT "ERROR, CHECK REGISTER IS FULL. HIT ENTER T
     O CONTINUE."
1040 GOTO 1190
1050 CR$(J,I)=WT$+CHR$(255)+STR$(AM)+CHR$(255)+NT$+CHR
     $(255)
1060 IF DW$<>"" CR$(J,I)=CR$(J,I)+DW$ ELSE CR$(J,I)=CR
     $(J,I)+LEFT$(TIME$,8)
1070 IF CC<>0 GOTO 1130
1080 IF (AB(I)-AM)>=0 GOTO 1130 ELSE PRINT "BALANCE WI
     LL GO NEGATIVE."
1090 INPUT "WRITE CHECK ANYWAY";AN$
1100 IF AN$="YES" OR AN$="Y" GOTO 1130
1110 IF AN$="NO" OR AN$="N" CR$(J,I)="" : RETURN
1120 GOTO 1090
1130 AB(I)=AB(I)-AM
1140 IF CN=-1 THEN CN=CN(I) : CN(I)=CN(I)+1
1150 CR$(J,I)=STR$(CN)+CHR$(255)+CR$(J,I)
1160 IF AB(I)<0 PRINT "BALANCE IS NOW NEGATIVE."
1170 IF OP<>1 RETURN
1180 PRINT "ACCOUNT POSTED. HIT ENTER TO CONTINUE."
1190 A$=INKEY$ : IF A$="" GOTO 1190 ELSE IF ASC(A$)<>1
     3 GOTO 1190
1200 RETURN
```

As is readilly seen, lines 910 through 1200 are more complicated than any we've used so far. Line 1050 deserves some special attention. Here we are concatenating (computerized glueing together) several strings of characters into a single character string. Each of the separate strings is delineated with a CHR$(255). This flag will be used to break the string apart for later processing. For now, it keeps everything about a check in a single entry that is easily decoded.

Attention should also be given to the technique used to find the account number (line 980). Here we have an example of jumping out of a FOR...NEXT look without completing the loop. This is normally *bad* coding technique. The reason we use it here is that we will not return to that loop once we've broken out of it. If we find what we're looking for, great! The variables are all set correctly and that's all that's needed. If we don't find the account number, we can then tell the user so that he can establish the account or supply the correct account number.

Now that the POST routine is in place it's a relatively easy task to write the WRITE routine. We simply call the POST routine after calling the common data input routine. We also need to use a special routine to format the check and print it. The thing that makes it

special is the conversion of numbers to words. In short, we must make a value like "$1253.22" come out as "ONE THOUSAND TWO HUNDRED FIFTY-THREE AND 22/100's" on the check. That may sound difficult, but it really isn't! Here then is the WRITE routine in it's simplest form:

```
1210 REM ------------------------------------------
1220 REM          WRITE A CHECK ROUTINE
1230 REM
1240 IF AN(1)=0 GOSUB 1860
1250 CLS
1260 PRINT STRING$(20,"-")" CHECK WRITING ROUTINE "STR
     ING$(21,"-")
1270 CC=0 : GOSUB 600  : GOSUB 980
1280 IF AN$<>"N" AND AN$<>"NO" GOSUB 2550
1290 RETURN
```

There really isn't much to it, is there? Only nine lines (three of which are REMarks). Fortunately all of the hard work is being performed in other routines. That's the reason there are three GOSUB statements. We'll look at the check formatting routine in a little while so just be patient!

Since we've now been able to write checks as well as post hand written checks, we should be able to print the check register. In order to do that we need to write the STATEMENT routine! See how easy that is? Anyway, here it is:

```
1300 REM ------------------------------------------
1310 REM          PRINT CHECK REGISTER
1320 REM
1330 CLS
1340 PRINT STRING$(21,"-")" PRINT CHECK REGISTER "STRI
     NG$(21,"-")
1350 PRINT
1360 HD$="CHK #  TO WHOM WRITTEN     AMOUNT           NO
     TES                  DATE"
1370 F$="#####  %                          %$##,###.## %
     %  %       %"
1380 D$="       %                          %$##,###.## %
     %  %       %"
1390 INPUT "FOR ALL ACCOUNTS";AN$
1400 IF AN$="YES" OR AN$="Y" GOTO 1460
1410 IF AN$<>"NO" AND AN$<>"N" GOTO 1390
1420 INPUT "ACCOUNT";AC
1430 FOR I=1 TO MA : IF AN(I)=AC GOTO 1470 ELSE NEXT I
1440 PRINT "ERROR, ACCOUNT NOT FOUND. RE-ENTER."
1450 GOTO 1420
1460 I=1
1470 LPRINT "ACCOUNT NUMBER: "AN(I);TAB(59);
1480 LPRINT "REPORT DATE: ";LEFT$(TIME$,8)
1490 LPRINT " "
1500 LPRINT "CHECK REGISTER:"
1510 LPRINT " "
1520 LPRINT HD$
1530 LPRINT " "
1540 CA=0 : DP=0
1550 J=1
1560 IF CR$(J,I)="" GOTO 1720
1570 CR$=CR$(J,I)
1580 FOR K=1 TO 4
1590    L=INSTR(CR$,CHR$(255))
1600    PT$(K)=LEFT$(CR$,L-1)
1610    CR$=RIGHT$(CR$,LEN(CR$)-L)
1620 NEXT K
1630 CN=VAL(PT$(1))
1640 AM=VAL(PT$(3))
1650 IF CN>=0 THEN CA=CA+AM : GOTO 1690
1660 LPRINT USING D$;PT$(2),AM,PT$(4),CR$
1670 DP=DP+AM
1680 GOTO 1700
1690 LPRINT USING F$;CN,PT$(2),AM,PT$(4),CR$
1700 J=J+1
1710 GOTO 1560
1720 LPRINT " "
1730 LPRINT USING "BEGINNING BALANCE:      $##,###.##";
     AB(I)+CA-DP
1740 LPRINT USING "DEPOSITS:               $##,###.##";DP
1750 LPRINT USING "TOTAL CHECKS:           $##,###.##";CA
1760 LPRINT USING "CURRENT BALANCE:        $##,###.##";
     AB(I)
```

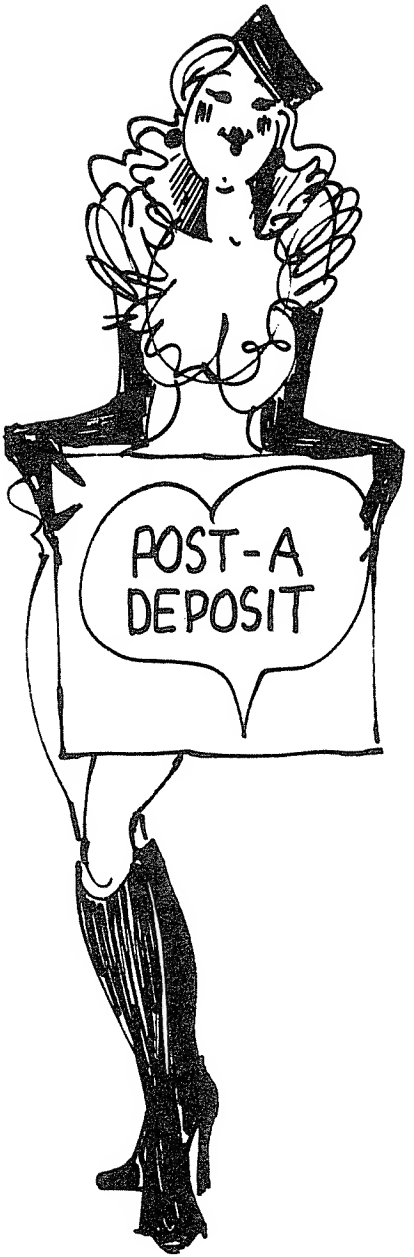
```
1770 IF AN$="NO" OR AN$="N" GOTO 1820
1780 I=I+1
1790 LPRINT " "
1800 LPRINT " "
1810 IF I<MA AND AN(I)<>0 GOTO 1470
1820 IF OP<>3 RETURN
1830 PRINT "HIT ENTER TO CONTINUE"
1840 A$=INKEY$ : IF A$="" GOTO 1840 ELSE IF ASC(A$)<>1
     3 GOTO 1840
1850 RETURN
```

The STATEMENT routine, which occupies lines 1300 to 1850, is by far the longest routine we've written so far. The reason is that it does more than any of the others! It must break the CRF entries apart (lines 1570 through 1620) as well as setting up the various print formats and so on. As is easily seen, however, the routine is not at all complex, it just takes more lines of code than the others. So far we've been able to test the program functions as we've added them, but we haven't been able to set up any accounts. That's for a reason. We've been testing the error handling routines. Here then is the routine used to set-up and account. It's short and to the point.

```
1860 REM ------------------------------------------------
1870 REM            SET-UP AN ACCOUNT
1880 REM
1890 CLS
1900 PRINT STRING$(20,"-")" ACCOUNT SET-UP ROUTINE "ST
     RING$(20,"-")
1910 PRINT
1920 IF AN(1)=0 PRINT "NO ACCOUNTS SET-UP YET."
1930 PRINT
1940 FOR I=1 TO MA : IF AN(I)=0 GOTO 1970 ELSE NEXT I
1950 PRINT "ERROR, ACCOUNT REGISTER FULL"
1960 RETURN
1970 INPUT "ACCOUNT NUMBER";AN(I)
1980 INPUT "STARTING ACCOUNT BALANCE";AB(I)
1990 INPUT "NEXT CHECK NUMBER FOR THIS ACCOUNT";CN(I)
2000 PRINT
2010 PRINT "ACCOUNT ESTABLISHED. HIT ENTER TO CONTINUE
     ."
2020 A$=INKEY$ : IF A$="" GOTO 2020 ELSE IF ASC(A$)<>1
     3 GOTO 2020
2030 RETURN
```

In lines 1860 through 2030 we simply get the account number (and make sure we have room for it in the array) and the starting balance. By the way, if you are dealing with alphabetic account numbers the array type will have to be changed from numeric, double precision to string. A simple change made by deleting line 50 and appending the string flag ($) to the end of the variable name (AN$)!

Since, from time to time, a person who writes checks might like to make a deposit to the checking account, we've thoughtfully provided a way to enter that data. In lines 2040 to 2230 there is a deposit routine which looks like this:

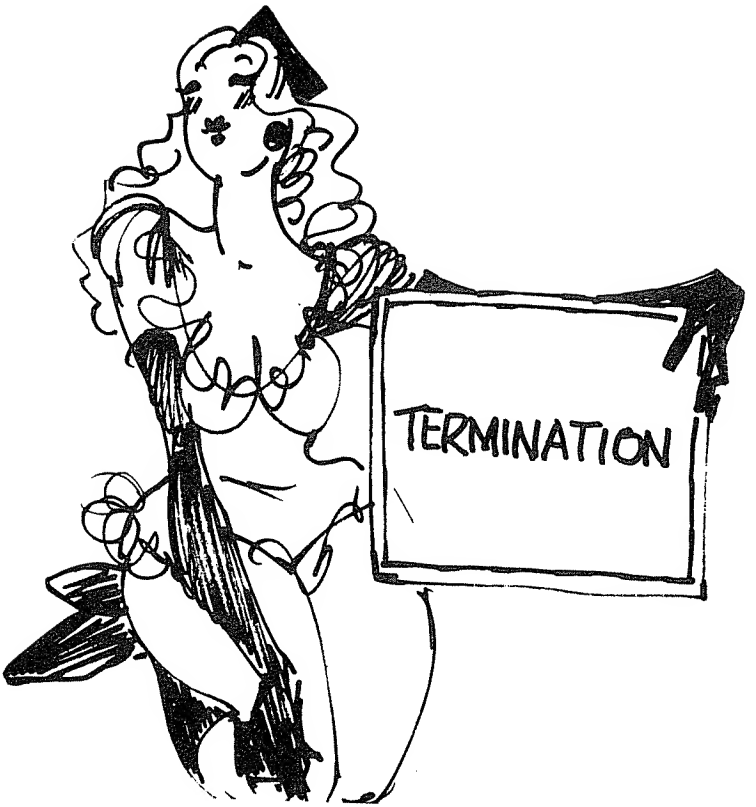



```
2040 REM ------------------------------------------
2050 REM                POST A DEPOSIT
2060 REM
2070 CLS
```

```
2080 PRINT STRING$(24,"-")" POST A DEPOSIT "
     STRING$(24,"-")
2090 PRINT
2100 INPUT "ACCOUNT NUMBER";AC
2110 FOR I=1 TO MA : IF AN(I)=AC GOTO 2140 ELSE NEXT I
2120 PRINT "ERROR, ACCOUNT NUMBER NOT FOUND. RE-ENTER"
2130 GOTO 2100
2140 INPUT "DEPOSIT AMOUNT";DA
2150 FOR J=1 TO MN : IF CR$(J,I)="" GOTO 2190 ELSE NEX
     T J
2160 PRINT "ERROR, CHECK REGISTER FULL. HIT ENTER TO C
     ONTINUE"
2170 A$=INKEY$ : IF A$="" GOTO 2170 ELSE IF ASC(A$)<>1
     3 GOTO 2170
2180 GOTO 2230
2190 CR$(J,I)="-1"+CHR$(255)+" "+CHR$(255)+STR$(DA)+CH
     R$(255)+"DEPOSIT"+CHR$(255)+LEFT$(TIME$,8)
2200 AB(I)=AB(I)+DA
2210 PRINT "DEPOSIT POSTED. HIT ENTER TO CONTINUE."
2220 A$=INKEY$ : IF A$="" GOTO 2220 ELSE IF ASC(A$)<>1
     3 GOTO 2220
2230 RETURN
```

As usual we get the account number and verify it before we actually post the account. Since deposits are fairly simple, all we need is the amount and the account number. We can then build the CRF entry using the current system date and the dummy note DEPOSIT. Actually, it might be nice to add the deposit date so that the register will accurately reflect the current account status. That enhancement is left to the reader. After all, we're supposed to be learning from this book, not buying a finished piece of software. There will be more enhancements mentioned as we go along.

Once everything has been done, it might be nice to write the modified CRF back to either disk or cassette before terminating the program. For that purpose we have the END routine. This again is a very short routine:




```
2240 REM ------------------------------------------
2250 REM         TERMINATION PROCESSING
2260 REM
2270 CLS
2280 PRINT STRING$(21,"-")" TERMINATION PROCESSING "ST
     RING$(22,"-")
2290 PRINT
2300 PRINT "WRITING CHECK REGISTER FILE."
2310 OPEN "O",1,"CRF/DAT"
2320 FOR I=1 TO MA : PRINT #1,AN(I) : NEXT I
2330 FOR I=1 TO MA : PRINT #1,AB(I) : NEXT I
2340 FOR I=1 TO MA : PRINT #1,CN(I) : NEXT I
2350 FOR I=1 TO MA
2360    FOR J=1 TO MN
2370        IF CR$(J,I)="" GOTO 2400
2380        PRINT #1,J,I,CR$(J,I)
2390    NEXT J
2400 NEXT I
2410 CLOSE
2420 PRINT "PROGRAM ENDED."
2430 END
```

Consisting of lines 2240 to 2430 the termination routine writes over an existing CRF or creates a new CRF with the data that is in the CRF arrays. As another enhancement, it migh be nice to be able to specify a file name in both the INITIALIZATION and END routines so that you can use any file name as opposed to just CRF/DAT.

Now the fun part begins. You may remember that we talked about the fact that we should be developing a library of subroutines. Well, as we were developing this program, I happened to remember a subroutine that had been floating around on one of my disks "somewhere" in the disk cabinet. Just exactly where, well I wasn't quite sure. After some searching (next week I just gotta get organized) I was able to find the routine I wanted.
In lines 2440 through 2510 shown here:

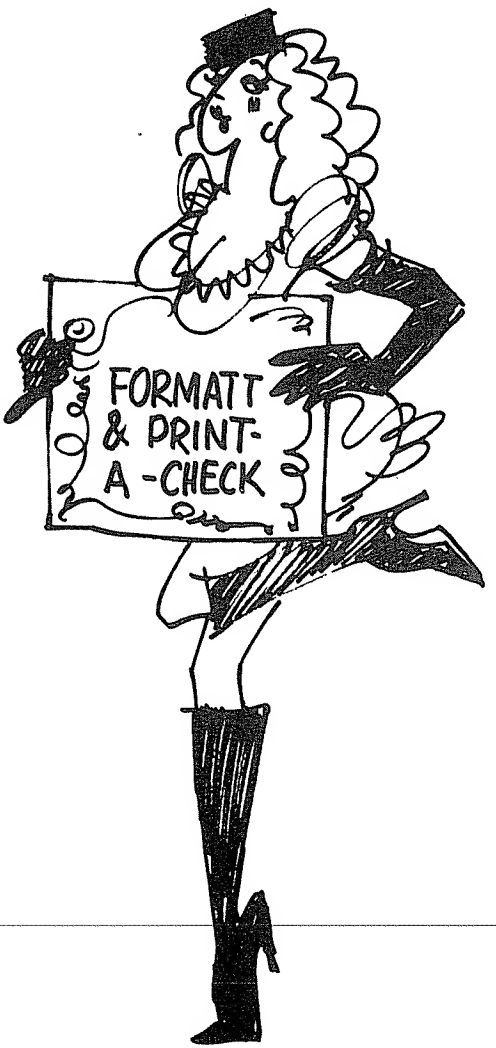
```
2440 REM ------------------------------------------------
2450 REM       FORMAT & PRINT CHECK (DATA)
2460 REM
2470 DATA ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,NINE
     ,TEN
2480 DATA ELEVEN,TWELVE,THIRTEEN,FOURTEEN,FIFTEEN,TWEN
     TY,THIRTY
2490 DATA FOURTY,FIFTY,SIXTY,SEVENTY,EIGHTY,NINETY
2500 DATA JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,
     DEC
2510 FOR I=1 TO 23 : READ NU$(I) : NEXT I : FOR I=1 TO
     12 : READ MO$(I) : NEXT I : RETURN
```

we see a rather cryptic comment "FORMAT & PRINT CHECK (DATA)" on a REMark statement. The data seems to be the names of some numbers, perhaps choosen at random. Actually, there is a specific pattern to these numbers. In fact, the names are for those numbers which are unique. The only exception is the number fourteen which could have been made up of the name "four" and the word "teen." The reason we didn't do it that way is because fifteen is again a unique word. All other values can be made up of the names here plus the words HUNDRED and THOUSAND. As you will see, these words are included in the code in lines 2520 to 2800 which we'll get to in a moment.

The remaining data is the abbreviations for the month names which are printed on the check in the place for the date written. Line 2510 actually loads these values into their respective arrays.

What is it we are trying to do here? We'll be taking the check amount and converting it to words since that is the way in which checks are written. Both numbers and words are usually included on the checks (to prevent altering the amount of the check). Usually this is omitted if a "check writer" is used since these machines will emboss the check with the amount making alteration exceedingly difficult.

Let's take a look at the code that we've used, and then we'll explain the logic behind it since we haven't covered it anywhere else in the book. Here's the code:



```
2520 REM ------------------------------------------
2530 REM         FORMAT AND PRINT CHECK
2540 REM
2550 WD$=""
2560 A1=INT(AM/1000)
2570 A2=INT((AM-A1*1000)/100)
2580 A3=INT(AM-(A1*1000+A2*100))
2590 A4=AM-INT(AM)
2600 IF A1<>0 THEN WD$=NU$(A1)+" THOUSAND "
2610 IF A2<>0 THEN WD$=WD$+NU$(A2)+" HUNDRED "
2620 IF A3=0 GOTO 2700
2630 IF A3<16 THEN WD$=WD$+NU$(A3) : GOTO 2700
2640 IF A3<20 GOTO 2690
2650 A5=INT(A3/10) : A3=A3-10*A5
2660 IF A3<>0 GOTO 2680
2670 WD$=WD$+NU$(A5+14) : GOTO 2700
2680 WD$=WD$+NU$(A5+14)+"-"+NU$(A3) : GOTO 2700
2690 WD$=WD$+NU$(A3-10)+"TEEN"
2700 WD$=WD$+" AND"+STR$(INT(A4*100))+"/100's"
2710 LPRINT TAB(41);MO$(VAL(TIME$));"";MID$(TIME$,4,2)
     ;",";TAB(56);MID$(TIME$,7,2)
2720 LPRINT " "
2740 LPRINT TAB(10);WT$;TAB(58); : LPRINT USING "**,##
     #.##";AM
2750 LPRINT " "
2760 LPRINT WD$
2770 LPRINT " "
2780 LPRINT " " : LPRINT " "
2790 LPRINT TAB(5);NT$
2800 RETURN
```

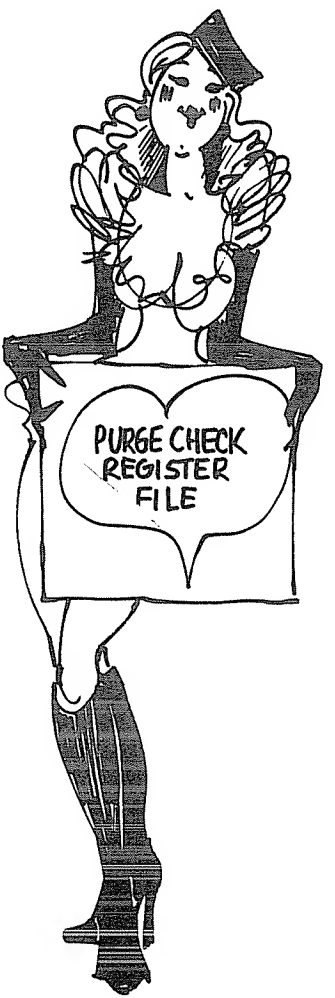
As you can see, lines 2520 through 2800 form the body of the routine. The routine itself is actually broken into four functional sections (mini-routines?). Lines 2550 through 2590 perform the "initialization" logic. The variable that will hold the final text (WD$) is set to a "null string" (a null string is a character variable that contains no data and has a length of zero), the amount field is broken down to facilitate table lookup (lines 2560-2650). This breakdown is accomplished by the careful use of the INT function. We strip off the part of the number that we don't want by dividing by the appropriate power of 10 (1000, 100, and 10) and then getting rid of the decimal portion.

Having accomplished this splitting apart, lines 2660 through 2700 actually format the text. The thousands and hundreds are very straight-forward for the amounts we are using (if the amount exceeds 15,000 we'll need to add some extra logic to handle the numbers in the same fashion as we handled the tens). The tens are another problem. If the number is between 1 and 15 we can get the value directly via table lookup, but if the number is between sixteen and nineteen we need to form a composite word (see line 2690). For numbers between 20 and 99 we need still more logic. If the number is an even multiple of 10 (20, 30, 40, etc.) then we need only the first part of the name (line 2670). However, if the number is not an even multiple we again must form a composite word (as in line 2680). Finally, the cents amount is multiplied by 100 and the decimal part (caused by rounding error) is truncated and thrown away. The result is then placed at the end of the string of words followed by the constant "/100's." As an enhancement here, we might want to place the word "NO" over this terminator if the cents value is zero.

Once the amount has been converted into words, lines 2710 through 2790 do the actual printing of the check. This is in the rather typical "personal check" format which is easilly changed for other check formats. Note that the date is extraced from the system variable TIME$ (on the TRS-80) so that the date can be printed in a more usual fashion, for example, "APR. 13, 82" where the check contains the "19" part of the year.

We are nearly through with the discussion of this version of the program. The only section that remains is the PURGE routine that will print a final report and then eliminate the data. This is done so that only the most current data is kept on file. Purges might be run on a monthly or quarterly basis depending upon how many checks are written and deposits made during any given time period. Remember, the program only has room for 150 entries before a purge becomes necessary (unless, of course, you change the dimension variable in line 60).

Here then is the PURGE routine:



```
2810 REM ------------------------------------------
2820 REM            PURGE CURRENT CRF
2830 REM
2840 CLS
2850 PRINT STRING$(22,"-")" PURGE CURRENT CRF "STRING$
     (23,"-")
2860 GOSUB 1360
2870 IF AN$="YES" OR AN$="Y" GOTO 2890
2880 FOR I=1 TO MA : IF AN(I)=AC GOTO 2900 ELSE NEXT I
2890 FOR I=1 TO MA
2900    FOR J=1 TO NM
2910        CR$(J,I)=""
2920    NEXT J
2930    IF AN$="NO" OR AN$="N" GOTO 2950
2940 NEXT I
2950 PRINT "CRF PURGED. HIT ENTER TO CONTINUE."
2960 A$=INKEY$ : IF A$="" GOTO 2960 ELSE IF ASC(A$)<>1
     3 GOTO 2960
2970 RETURN
```

Comprising lines 2810 to 2970 this is the last routine in the program. It uses the STATEMENT routine to format and print the reports and then resets the CRF array to nulls. This does not affect the account number, account balance, or check number arrays which must be left untouched.

This is, of course, only a basic program. There are many enhancements that can be made to make the program more useful. For example, you might want to consider adding a routine that displays the account numbers and allows the user to select a number associated with that account rather than requiring the entire account number to be input for each check posted or written and for each deposit.

Another enhancement might be to add a comment for each account (personal, business, petty cash, etc) that could also be stored. If there are multiple checking accounts this would certainly make the selection of an account easier.

You will notice that there are two routines that have not been implemented. One of these is the posting of interest (for accounts that get interest). This is left to the reader as an exercise. It's not a difficult enhancement, but it's one that will definitely test your understanding of the material we have presented so far.

Another routine that could be added would be a way to post service charges. With the program as it curently stands, you could post interest charges as a hand written check using check number 1 (or some non-zero check number reserved for that purpose). Again, adding this routine is not difficult, and will give you a chance to play with a program that is fully debugged and functional as it stands. After all, that's the sort of thing that you'll be doing more and more of as your routine library gets bigger and bigger. With more routines the development of new programs will become less and less tedious and more and more a function of modifying an existing routine or program by adding or removing features.

That's life in the fast track! All programs lend themselves to that sort of development and the nice thing about it is that it makes your job as a programmer so much easier. Just think, it's almost as if you had the computer writing your programs for you! In fact, it's one step

better, you are using your computer as a tool to assist you in developing a program rather than just as a passive device being banged upon as you press the keys writing the "definitive" sort one more time!

This Chapter has shown us one way in which the program could have been developed, a modular approach. It was fairly easy to see the modules in the program (especially since we broke it out for you). By the way, the complete listing is also available in the back of the book in Appendix A.

As we go on to the next Chapter we'll be taking another look as this program. This look will focus on a way in which we could have used the structured techniques to write the same program. Rather than rewriting the program (which serves little purpose) we'll look at the structural differences that would be necessary, and will take some of the routines and write them in a structured fashion so that you can get the flavor of the technique. In Appendix B is the fully rewritten program so that, for the curious, you can see how it might have looked.

One last point here, this is the way in which I have written this program. Obviously other people might have approached the program from an entirely different perspective and therefore produced an entirely different program that performed the same functions. That's perfectly natural and good! In fact, I'm all in favor of that. Variety is, after all, the spice of life! The important point here is that just because I've done this program in this fashion doesn't make it right (or wrong). This is my style, and not necessarily yours. If you want (and I do recommend this) write the program your way! In doing so you will be enhancing your own creativity as well as testing the material in this book.

CHAPTER 9

Structured Approaches

The word "structured" brings all sorts of visual images to mind; it implies an underlying organization, something that is composed of parts. This is especially true of "structured programs." In this chapter we will be taking another look at the CHEKBOOK program in an effort to evaluate the differences between the modular approach to writing a program and the structured approach.

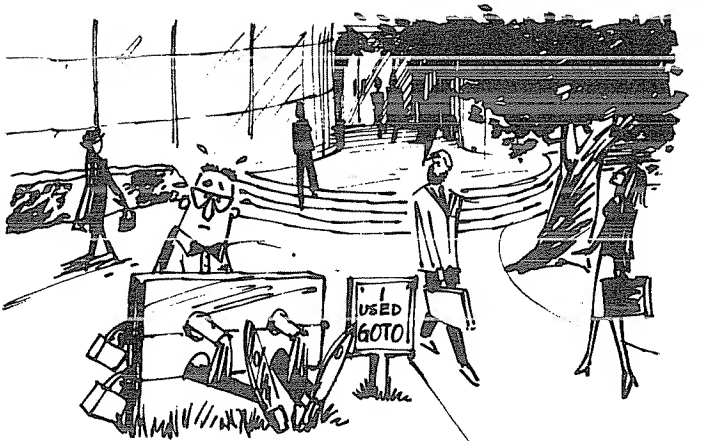
Let's begin by reviewing some of the history behind the structured approach so that we can place it in perspective. With this understanding, we'll be able to view the topic in a clear and unemotional fashion (And if you don't, I'm gonna beat your computer into scrap iron!). Actually, that untypical tirade (I'm usually quite calm) is used to point out that many of the proponents of this method are quite obviously bigoted and don't understand the methodology well enough to see that there may be many valid ways of viewing "structured programming."

Professor Edsger Dijkstra first coined the term "structured programming" sometime in the mid-1960's. In 1972 a book called "*Structured Programming*" was written by O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare (published by Academic Press, New York). The book consisted of three separate notes dealing with "Structured Programming," "Structured Data," and "Hierarchical Program Structures." In this book the methodology was fully defined which would later lead to a large number of enhancements to the fledgling science called "Software Engineering."

This methodology, briefly stated, held that there must be an inherent logical structure, not only for the program but also for the design effort and the data processed by the program. The actual implementation of these ideas waited for others to build upon them and make them real. The first such test came about in the early 1970's when IBM (whoever they are) undertook a project in New York for the New York Times. The structured approach (as it was then envisioned) made this monumental task one of relative ease. The project was finished ahead of schedule and under budget. IBM, which was notorious for good management practices of budgeting and scheduling, was amazed!

In the various trade journals the phenomenal success of this technique was praised and discussed. Unfortunately, there were several innovations that IBM was using. One of these was the Chief-Programmer approach wherein a "super" programmer does most of the work assisted by mere mortal programmers. These lesser beings actually wrote much of the code and performed the more mundane tasks of library maintenance and so on. This freed the creative genius of the super programmer for more productive work.

In any event, the various "software houses" and consulting agencies looked at the final product and began designing courses explaining the so-called "structured" methodology as each of them perceived it. Unfortunately, these perceptions were often incomplete and incompatible!



This incompatibility has led to the running argument of the presence of the "GOTO" verb in a program. Some will claim that it should never be found in a program while others claim that it may be used "only in dire need." Still others (myself included) say it can be used where it is the most obvious verb to use provided that its use will not reduce the readability and maintainability of the program.

Another problem that has developed as a result of the disagreement over "GOTO" is the use of nested IFs. To define this, let's take an example. I should warn you that not all of the various BASIC dialects available on the popular microcomputers today will allow nested IFs.

Let's assume that we want to perform some action based upon a number of conditions. The actions are as follows; if the variable A is equal to 5 and the variable B is greater than 1 we will terminate the current processing (via a RETURN) but if A is equal to 5 and C is equal to 1 we will set B equal to 9. We could code this as:

```
IF A=5 AND B>1 RETURN
IF A=5 AND C=1 THEN LET B=9
```

or we could write:

```
IF A=5 THEN IF B>1 RETURN ELSE IF C=1
        LET B=9
```

Both of these statements say the same thing. If A is not equal to 5, neither of the other tests will be made, while if B is greater than 1 we will RETURN otherwise the test for C being equal to 1 will be made.

The obvious complaint is that this code can become quite hard to follow if there are many levels. One solution that has been proposed is to indent each level of IF so that the current level is visually obvious. Taking our last IF this might look like:

```
IF A=5 THEN
     IF B>1
        RETURN ELSE
     IF C=1
        LET B=9
```

This makes the levels (2) very obvious in exactly the same way as the levels of pseudo-code were obvious. In fact, this is always a good idea. Anything that you can do to make a program easier to read will more than likely make it easier to maintain later on!

If we forget about all of these petty arguments (and most of them are) we will be able to concentrate on the more important aspects of this particular methodology. Why would we want to use structured programming? What are the basic benefits? Who cares, anyway?

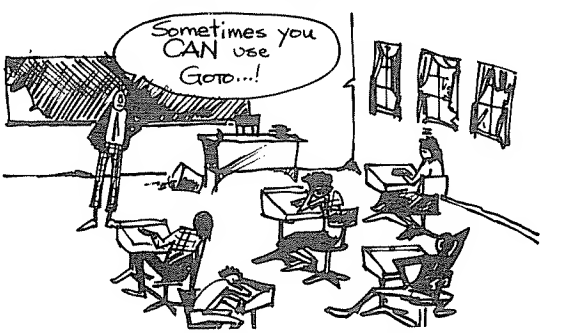
Now, let's take a look at these questions. By answering them we should be in a better position to evaluate the usefulness of this technique. With regard to why we might want to use structured programming there are many possible answers. The most obvious answer is that we might *not* want to use it! I'll bet you didn't expect that answer, did you?

Actually, the statement may be true! Assuming (for the moment) that you do want to use structured programming we can evaluate the reasons. Structured programming allows us to write a program in a step-wise fashion, that is, we can write each section of code from the highest level to the lowest as we need it. There's no need to consider routines outside of the one we are currently using.

"But," you say, "that's not too different from modular programming!"

Very good, you've been paying attention to the material as it's been presented! As it stands, there's very little difference. However, the major difference is the way in which the routines are "entered." By "entered" we mean the transfer mechanism (like GOTO or GOSUB). With structured programming a routine is *always* entered via a GOSUB and exited via a RETURN. There should only be one exception: the termination routine may exit with an END statement (STOP should not be used except during the debugging stage since it is possible to CONTinue from a STOP).

Another major difference between modular programming and structured programming is the main line. For modular coding it may be quite lengthy. On the other hand structured programming allows a very short main line — it should only contain the GOSUBs needed to transfer control to the appropriate subroutines. There may be a GOTO at the bottom of the main line that will return control to the top of the main line (unless your particular dialect of BASIC will support a WHILE...WEND type of construct).



Now that we've addressed the reasons for using structured programming, let's take a short look at some of the benefits. As we saw with the modular approach to the CHEKBOOK program, there were a few problems in testing since the needed routines (such as SETUP) weren't coded until later in the program coding cycle. With structured programming this difficulty is overcome. Since the amount of pre-definition time is greater for structured programming than it is for modular coding there will be less chance for an oversight of this nature.

As you may have guessed, the oversight was deliberate, and was added for several reasons. Let's take a look at the main line of our structured program:

```
50 REM ------------------------------------------------
60 REM                MAIN LINE CODE
70 REM
80 GOSUB 120                        'INITIALIZATION
90 GOSUB 590                        'MENU
100 ON OP GOSUB 800   , 1040 , 1250 , 1790 , 1980 , 217
    0 , 2350
110 GOTO 90
```

As we can see, there are only four lines of code, two GOSUBs, one ON...GOSUB, and one GOTO. We're not counting REMarks statements since that would up the count to 7 lines, a truly excessive number! Note also that the initialization routine which came first in the modular version is now contained somewhere else in this version of the program. The only exception is the CLEAR statement which is used to reserve string space. If this had been included in the initialization routine it would also have cleared the GOSUB return pointer so that we would have had an error (RETURN without GOSUB).

This initialization routine actually takes all of the code from the other version of the program and places it all in one central location so that all initialization is performed at one time; then the routine is no longer needed.

As you look through the listing of this program you will find an occasional GOTO contained in the code. This is because I tried to maintain the same routines as were used in the modular program so you could see the similarities between the two programming methodologies. Another reason for the presence of the GOTOs is that I'm not a purist, I am not convinced that GOTO should be abolished!

Are there any other benefits that might be accrued as a result of using structured programming? Well, it won't make you sexier or more popular, nor will it keep you from getting cavities and early morning pasty film. What it will do is place you in the main stream of programming philosophy. Since structured programming is the current "fashionable" technique you will be more saleable if you are familiar with the style. If you aren't selling your talents (as a programmer) you might still be interested since the bulk of the trade journals are spending considerable time discussing this technique. The more conversant you are with structured programming the more you will be able to get from these discussions and the more you will be able to enhance your own programming abilities.

We did have one last question to address, "Who cares, anyway?" Remember that one? Actually, you should care! The purpose

here is not to convince you that structured programming is the best thing since buttered popcorn (I don't think it is), but rather that it is a technique with a lot of things to recommend it. It reduces the amount of time spent programming; it provides cleaner, more readable source code (programs); it minimizes unit test time (time spent debugging the individual programs of a system).

But, this may all seem very academic. What single reason is there for using (or not using) structured programming?

That's a hard question to answer. There are really so many different parts to that question that it would take a book all by itself to answer (and maybe even that wouldn't be enough). Instead, let's take a quick review of the program development cycle in terms of the amount of time spent. The table below summarizes the time allocations used before and after 1980 (an accepted before/after time frame for structured programming).

Activity	Pre-1980	Post-1980
Requirements	10%	10%
Feasibility Study	5%	5%
Design	20%	40%
Coding	20%	15%
Testing	40%	25%
Documentation	5%	5%

As you can see, this is rather different. The time allowed for design has doubled while the time for coding has gone down by 5% in spite of the fact that the current applications (programs) are far more complex than they used to be. The testing time has also been reduced as a result of the cleaner programs which resulted from better design. Actually, the Post-1980 figures are a bit optimistic and usually the design time comes out around 30% with the other 10% divided between coding, testing, and documentation.

In his "Notes of Structured Programming," Dijkstra points out that the size of a program is the very reason for a discussion of structured programming. We would certainly not attempt to write our Area Calculation program using the structured techniques because the overhead involved in building the modules and the linkage mechanisms between the modules would get in the way of the logic of the program. Structured programming is really a valid technique for moderate to large programs (however you measure that).

As was pointed out in Chapter 6 on flowcharts, structured programming really only uses three flowchart symbols; the "SEQUENCE," "IFTHENELSE" and "DOWHILE" symbols. During the mid-1960's two Italian computer scientists (Böhm and Jacopini) proved that these three symbols were the only ones needed and could ALWAYS be used to express any given flowchart. This provided a boost to the use of structured programming. It also made the design of programs easier by reducing the number of logical structures that must be considered.

Although this may sound rather far from the realm of microcomputers, it really isn't. Remember, structured programming was designed for moderate to large programs. It is programs of these very sizes that we have problems with on a microcomputer (either because of memory constraints or from language constraints).

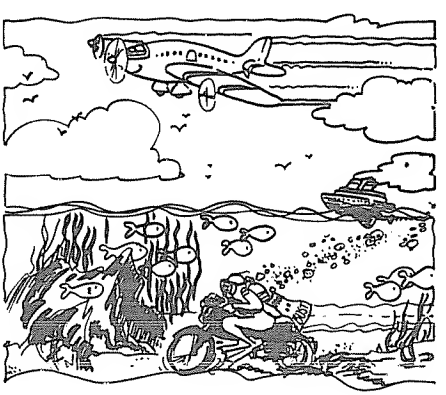
As we continue to enhance our own skills on the computer we own, we probably amaze ourselves at what can be done. Remember what it was like when the machine was new? For you newcomers, this should be encouraging — there's always a way to make the machine do what you want. Either through new skills or new approaches or...

Let's go back to the CHEKBOOK program and look at it with an eye to the differences caused by structured programming. The first item we see is that the hierarchy of routines (levels of importance) is more clearly carried out. That is, the higher level routines occur first while the secondary or lower level routines occur further

down the program. By the way, the term "down the program" simply means that part of the program with higher line numbers. It actually refers to a part of the listing which, if held vertically, would be at the bottom of the listing.

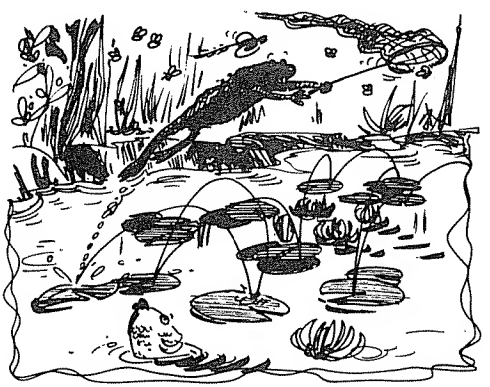
Interestingly enough, there are 64 GOTOs in this version of the program. It would certainly appear that I am not a purist! When compared to the modular version of the program we find that we did use fewer GOTOs (but only by 3). So, there must be something else about structured programming than the omission of GOTOs if I have guts enough to call this a structured program. Actually, I suffer from a problem similar to that which Dijkstra faced, a size restriction. If we were to develop a program of sufficient size to warrant the full use of all of the structured programming techniques, it would be too big to be used as an example in this book (but not too big for a microcomputer).

Perhaps a better way of facing this problem is to give some simple examples. We are often faced with the difficulty of expanding from a specific example and making a generalized statement out of it. As anyone who has taken a class in logic knows you can never go from the specific to the general, only from the general to the specific! Therefore, we are doomed to failure from the beginning. It's like taking the example we used earlier and saying an airplane and a motorcycle are the same thing because they can both be used to transport an individual from one location to another. Let's see a motorcycle, on its own, take someone from New York to Paris (France). It's a wet ride!



This is similar to the problem facing us when explaining structured programming. A program that is simple enough to be used for an example is too small to be conclusive proof for the advantages of structured programming! While this may seem to be self-defeating, we can approach the problem from a slightly different angle. If, instead of trying to find reasons for using the technique, we looked at the problems that face us when writing programs and look at the solutions that are offered by structured programming, we will have a better handle on the problem.

Ok, what are the problems? Complexity is the first problem. As a program gets bigger and bigger it becomes more complex. Debugging is another problem. We often wait until we've written the program before we begin testing and then we find that we've omitted parts of the program. As we add more code to compensate for the omissions the structure of the program starts to decay, and becomes less obvious. This, of course, makes the program more complex and therefore harder to debug, and so on, and so on,...



Documentation is always the bane of programmers. How does one go about documenting a large, cumbersome program? What is the minimum that is needed to allow use and maintenance of the program? How does one maintain the documentation as changes or enhancements are made to the program?

We'll tackle the documentation problem briefly here, and let the rest remain for Section 6. Can structured programming help us with documentation? Surprisingly enough, the answer is "Yes it can!" By reducing the documentation to the level at which you are working the amount of documentation is minimized (at any given point in

time), and the final result is synchronized with the program. This will make maintenance easier to perform and the documentation will be easier to update since it will already correspond to the structure of the program. More on that later.

Now we'll talk about the problems with coding and complexity. Let's address just one of the problems. When we were discussing the various "traditions" behind the different languages (back in Chapter 4) we mentioned that BASIC programmers had a tendency to place code anywhere they wanted in the program. This often resulted in GOTOs that jumped all over the place. Well, although I'm not a purist, that's the kind of GOTO I'd like to stamp out!

Structured programming is often cited as the chief cause of the discussion revolving around the existence of the GOTO verb. Actually, good programmers have always disliked the indiscriminate use of that particular verb. For example, imagine the following code:

```
120 A=B
130 GOSUB 1350
140 GOTO 900
.
.
900 C=B
910 GOTO 320
```

You can readily see that if the program consisted of a lot of code like that stuff above it would be very difficult to follow the logic. Remember the flowchart symbols that were used with structured programming? They all had a single entry point and a single exit point. They were, in effect, black-boxes that could be called upon to perform a given function without regard to how that function was performed.

If we build programs using the concepts of structured programming, we will have little difficulty in overcoming the problem of complexity. As someone once pointed out, God took six days to create the world (Genesis 1:1-31). Although I doubt that the writers of Genesis ever imagined they would be used as an example for good programming style we'll use them anyway. The idea here was to break a complex task into separate parts and tackle each individual part. The only problem was making sure that there would be proper interfacing. Such interfacing might be considered the separation of the water from the dry land, or the night from the day. In the case

of water we have the beaches, for night and day we have sunrise and sunset (which also makes a neat title for a song).

The key to structured programming is to eliminate unnecessary GOTOs that lead to different parts of the program. Within any given module or unit a GOTO may be used if it makes sense, and if it doesn't reduce the visibility of the logic. Therefore, we use GOSUB...RETURN to control the execution of the various components of the program. By the way, you will notice that I used the term "module" to refer to one of the components of a program; that's perfectly correct and natural. What we want to stress here is the idea of using *logic* to direct the formation of a program rather than necessity. By that we mean the program should be designed carefully enough that we won't have to code around missing logic in the design. Such coding can only lead to difficulty.

Ok, now that we've talked about the "theories" of structured programming, how do we go about putting it into practice? Actually, that's a lot easier than it may seem. We begin (as usual) at the top (I'll bet you thought I'd say beginning).

Just where is the top? If you'll remember the various discussions we've already had regarding the "dummy module" you'll be well on your way toward figuring out the top. We'll start by coding the highest level routines first and add the lower level routines once we've tested what we've already written. For example, we might first code the statements that identify the program, both by name and by function. This is then followed by the highest level routine, the main line.

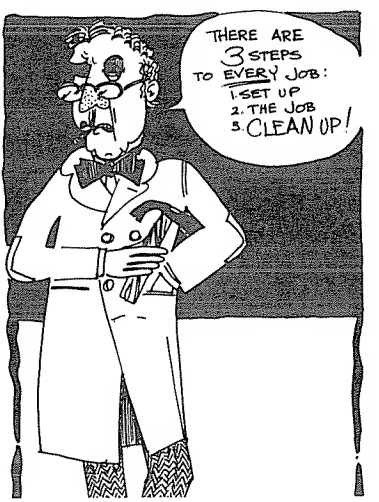
At this stage we will actually spend some "wasted time" by writing some code that will eventually be thrown away — RETURNs that correspond to each of the lower level routines that are invoked by the main line. We will not worry about routines that would be invoked by routines invoked by the main line! Everybody got that? We work at one level and dummy out the next level with RETURN so that we can test the code that's already been written. This allows us to overlap the test and coding cycles thereby shortening the time for both!

Having tested the main line (all four lines in the case of CHEKBOOK) we can then proceed to write the lower level routines, one at a time. Since there's still a question of sequence we look for some sort of indicator that would tell us the order in which they should be written. Usually there's an implied priority even within routines or

modules at the same level of the hierarchy. For example, we would probably write the MENU routine next since that is the key for getting to any of the other routines! Make sense? Of course it does!

No it doesn't! The next routine that should be written is the initialization routine. We've got to get all of the variables initialized before we can begin any actual processing. Way back in Jr. High School I had a Shop teacher who stressed that there were three stages in any job: setup, the job itself, and clean-up. In writing a program to perform a task we must remember those same three stages. Initialization can be considered the "setup" while termination is the "clean-up."

Once we've coded the initialization routine we can then proceed to the MENU routine. This is where the testing gets tricky. How do we know that the initialization proceeded according to plan? There's



two different approaches to this evaluation. The first is to examine every possible variable with some kind of automated approach. Such a program would have to be loaded into memory before the program is actually run since most microcomputer BASIC interpreters have the nasty tendency to reset all variables if you edit the program (which includes adding new lines to the program).

A second approach is to examine the variables that you *know* are used to see if they are correctly initialized. The drawback to this approach is that you must check not only the variables initialized in this routine but also those used in the other routines. This is not

necessary if the RUN command causes the interpreter to reset all variables. By the way, most of the computers will reset the numeric variables to zero and the string variables to null (strings that contain no characters and have a length of zero). We'll talk more about this in Chapter 10, Approaches to Testing.

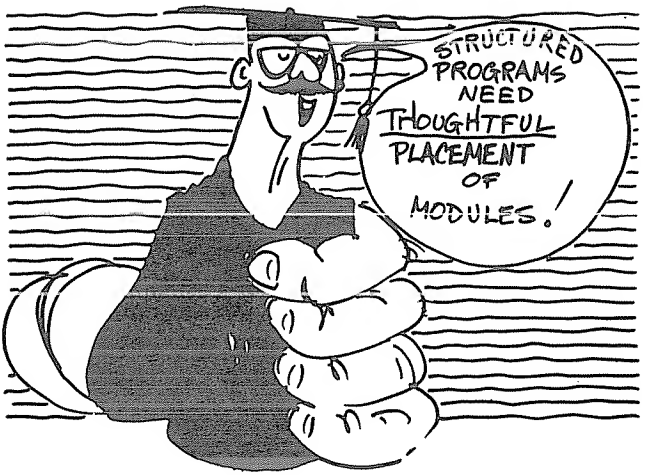
Once the MENU routine has been coded the program will be able to provide access to the other routines. At this stage we should look at any other ways to determine the coding sequence. Several ideas quickly come to mind. First, we decided that some of the routines (WRITE, PURGE) would use code that exists in other routines. Because these routines are now closely related they should be kept together so that their interdependencies (big word, means to rely upon the other) will be both logically and visually obvious. Therefore, if we choose to write the POST routine first, we would then take the WRITE routine immediately afterwords.

Another way to determine the sequence is to look at the numbers in the MENU and write them based upon that sequence. By the way, I chose not to place PURGE near STATEMENT for a simple reason. I felt that the use of the STATEMENT code was merely a convenience for PURGE. The primary purpose of PURGE was not printing a statement, but rather purging the CRF. The structure I chose was based upon the sequence of the routines in the MENU. Since that clearly points out a structure that is self-documenting I went with it. After all, it's much easier if you go with the flow!

This also has another nice property; the GOSUBs in the main line refer to increasingly higher line numbers as the MENU reference number gets higher. This seems natural to the orderly mind, and is therefore accepted more readily by another person who might, for some unknown reason, want to tamper with the perfection of my code. (Can you stand it?)

As we are all aware by now, there's more to programming than simply scribbling some lines of code on the back of a grocery bag with a crayon! If we have carefully thought out the program in advance, and have written it with the same care, there should be very few bugs. One axiom of programming that I fear we will never change is that there is *always* one more bug! Just because it hasn't been discovered doesn't remove the fact that it's there.

Now, having designed a good program, having written a good program, what's left? Well, we've still got to test the program and then we've got to document it. Each of these items will be covered



in some detail in the following chapters. At this stage it's important to note that we've now set the stage for what will follow. We've looked at two different ways of coding a program. Modular coding required a little less thought in terms of physical structure, and just as much in terms of how to break the program into modules.

Structured programming required a little more thought in terms of the placement of the modules and in terms of the flow of data through the program. This is the reason for the so-called structured data flow charts. We've not covered data too much, but that's because the study of data structuring is a very complex subject best left to the experts. There's no reason we need to be concerned with limited redundancy, independent structures, flat-files, inverted-files and all of the other buzzwords dropped so easily by those involved in data base management and data management.

Hopefully we also realized that the actual coding of a program is, in a sense, incidental to the actual task of "programming." It is merely the implementation, the embodiment, of the ideas which were carefully thought out in the design stages. I think that we've all come to realize that there's more to programming than just sitting at a keyboard and churning out a best-selling piece of code.

One thing that we've only touched lightly on is *what* to write. I suspect that the world really doesn't need a new version of LIFE or, perhaps, even STARTREK, regardless of how sophisticated. Games are fine and have their place, but the efforts that are put into game writing must be done in an area that will reap some benefits (other

than just enhancing your programming skills). Adventure games are really selling well (as of the time of this writing) and are also ideal areas for either modular or structured programming. Although I mentioned that we wouldn't be discussing games, I felt that this mention was justified based upon the popularity of certain kinds of games and the applicability of the programming methods we've just been studying!

Let's consider a few last points before we press on to the topic of testing. On several occasions we've mentioned that you should be building a library of routines for use when writing programs. As each problem is solved it can be cataloged. Later on, as you write other programs, you will be able to refer to it and save coding time by extracting fully functional and tested routines. This will obviously shorten not only coding time but also testing time.

As each routine is added to your library you are effectively adding "working capital" to your skills as a programmer. You will never again have to solve the same problem two, three or more times. The solution will be there, waiting to be used. If you continue to use either the modular or the structured programming techniques it will be easy to merge the routines into later programs because they are written in a data independent fashion. By that we mean that they only need to know a little about what is happening in the rest of the program. What little they do need is passed to them in the form of parameters and then acted upon.

The only changes that might be needed would be in the area of variable names, and that's where the idea of standardized names comes in handy. If certain variables or ranges of variables are always reserved for use in modules, there will be minimal overlap. All good ideas, what!


SSSHHHH!!!

Section 5 — Testing/Debugging

Now we've progressed to the point where we need to make sure the program does what it was designed to do. We do that through a process called "testing." The removal of problems or errors from a program is called "debugging" and we'll be looking at some easy ways to do that.

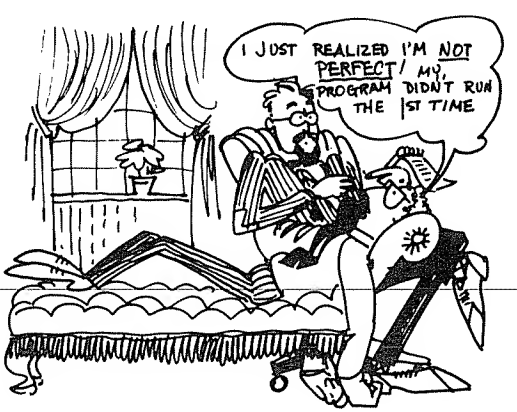
Throughout this Section we'll constantly be referring to the correct approaches to testing, and pointing out the areas where popular myth has misplaced the emphasis on testing.

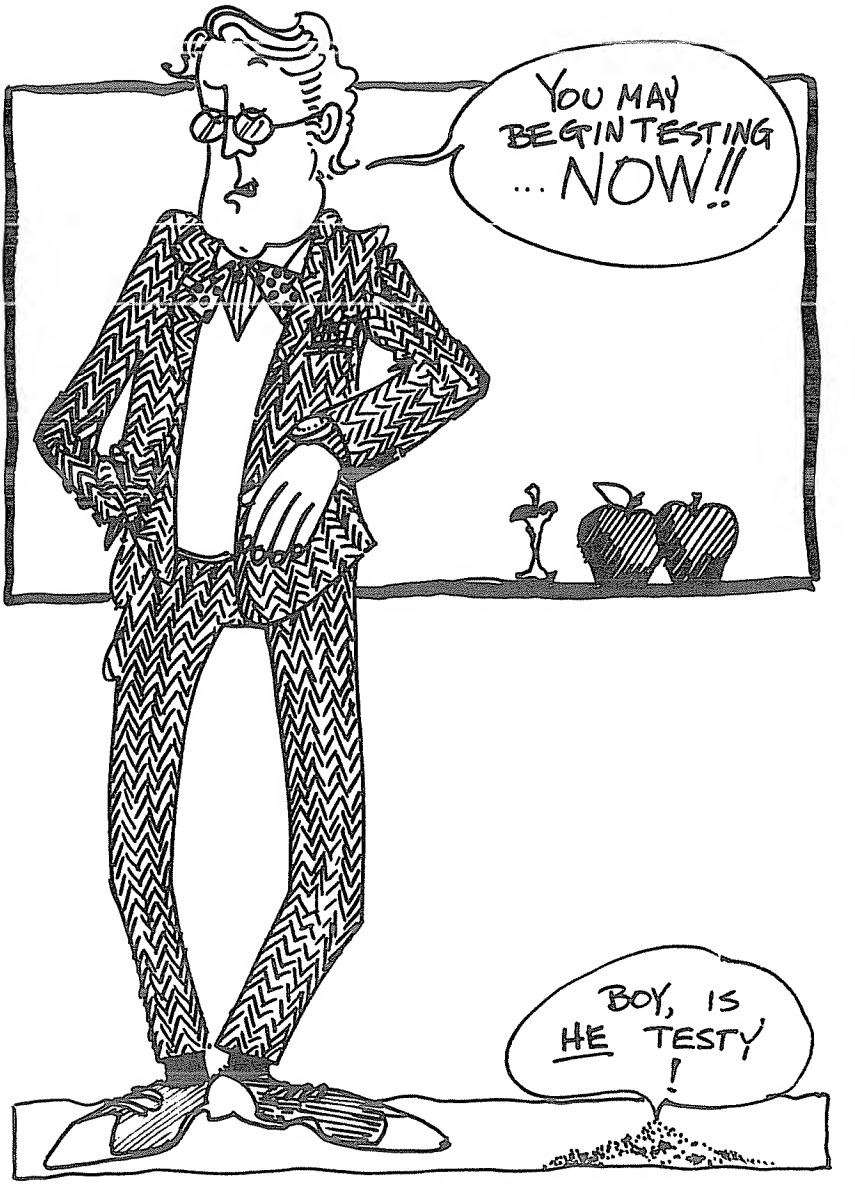
We've seen how good design can lead to simplifications in the program writing or coding stage, now we'll see where proper design of the tests can lead to simplification in the final checkout stages of a program's developmental life cycle. This is the area where most programmers are the weakest. This weakness may have several different causes and we'll be examining the various things which might reduce your effectiveness as a tester.

CHAPTER 10

Approaches to Testing

The idea of testing a program may be, to some, a nasty idea. After all, we've spent some time designing and writing the program, and there is no reason to assume that we're anything but perfect. If that is the case, the program should work correctly the first time! Right? Don't forget about Murphy!





We must all remember that even though we are perfect, there may be times when we are coding our programs that we aren't quite up to par. It's at times like that when we might let little errors creep into our code. If we can all admit that we have, on occasion, failed

to have a program work *exactly* the way we wanted it to the very first time it ran, we'll be well on our way to being good testers and debuggers. To see what that'll take, let's jump right in with a discussion of testing.

We could have called this chapter "Testing, Its Goals and Purposes" because that's exactly what we're going to be talking about. (The only reason I didn't call it that is 'cuz my publisher said it didn't sound very good!) Before we can discuss how to go about doing this thing called testing, we need to understand what it is that testing does and is.

Let's stop and think for a minute. When you have a friend over and you've just finished working on a program (assuming it's that kind of friend) what happens? The friend will usually say:

"What'll we do next?"

To which you reply "Let's see if it works!"

Do you agree? Well, you're doing it wrong! By the way, rule 1 of writing says never tell the reader he's wrong, but I believe that you're reading to learn, and to learn sometimes means unlearning bad habits.

Testing. The process of determining if a program will perform the functions it is supposed to perform without producing errors, either detected or undetected! The first assumption that you must make is that there *are* errors in your perfect program (which is, therefore, not perfect). Once you've made that assumption you can then proceed to the real business at hand, that of removing the bugs.

Why is this so difficult? Actually, it isn't. The major problem comes about when a reasonably intelligent person (you) is suddenly confronted with the possibility that you blew it! Right there, in front of your friend, is the evidence of your failure! Don't take it so hard, there's hope for you still. Actually, most programs don't work right the first time (an exception is this one: 10 END).

The psychological make-up of most programmers is such that there is often a problem in tearing their own code apart. There are two basic symptoms; either the code is right and the computer is wrong (possible, but not overly likely), or the programmer will rewrite entire sections of code in an effort to kill the bug by the sheer weight of lots of lines of code! Both approaches are a complete waste of time.

If we begin with the assumption that we are going to make the program shrivel up and die as a result of our testing we'll be way

ahead of the game! To get right to the point, let's take a couple of rather typical statements about testing a program. We'll examine each in turn to see how, if at all, they apply to the real world.

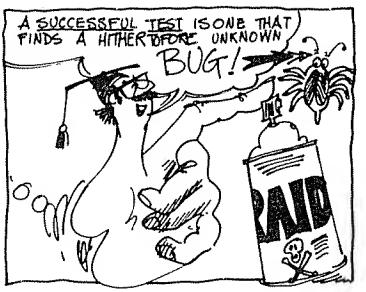
First, we can say that we test a program by making sure it does what we've asked it to do. Ok, fine! What have we asked it to do? Is the definition of the problem sufficient to allow the development of tests adequate to prove that the system will work? Will we be attempting to prove that a program works merely by excessive example (which isn't proof, but we'll come to that). Let's go back to our simple area calculation program. We know that we must test it to see if it will detect a length of zero, both length and width of zero, and also width and length equal to each other. Very good. Should we test other combinations to see if the program will fail if fed real numbers? What if the width is 9×10^{99} and the length is $9x10^{99}$ (which is a HUGE square)? Will the program correctly detect that? NO! The numbers are too large and will result in the program ending with an OVERFLOW error.

Should we prove that the multiplication routine in ROM will work correctly? If so, for how many cases should we prove it? That leads us to the second approach which says that we should test *every* possibility. Can you imagine testing a chess playing program for *every* possibility? I can't even imagine living that long. For our area calculation program we would have an infinitely long period of time if we tried every possible number. Even if we limited the possibilities to the real number scale and chose only integers it would still take more than my lifetime to complete! Actually, we'd suffer OVERFLOW long before we could ever test every possibility.

So, we can't test just for what is expected, and we can't test for every possibility. Just what the heck do we test for? It all depends! As an author I can get away with that statement, but it's really true. It depends upon the program, the amount of time available for testing, the criticality of the testing, and so on. The more critical the program is the more completely it should be tested. Imagine writing an operating system and releasing it to the public without thorough testing! IBM (those guys again) successfully did that for years; they'd let the customer find the bugs and assist with the fixes. Big companies can (sometimes) do that, micro users can't. We need to have a system work correctly the first time and every time.

For BASIC users some of the testing problems are quickly resolved. The interpreter will not allow string data where numbers

should be (but will allow numbers where string data should be). Part of the testing is therefore simplified. But there's another, nastier side to BASIC. Because of its ease of use, BASIC will often mask bugs. We can hit BREAK (or ESC or whatever) and stop the execution of the program, make a fix and continue. But, will we remember what that fix was? Will we care how it affects other routines? That's where the greatest amount of caution must be taken.



How, then, do we approach testing? We've already started by accepting the fact that the program contains one or more bugs. We next decide that the testing plan (yup, we writes test plans too) should have been laid down as part of the *design* effort. While a program is being designed is the best time to decide how it should be tested. In fact, one of the early concepts associated with top-down design was that you *start* by designing the test plan and work down until the program is in production! There's a lot of good to say about that approach.

But where does that leave us? Most of us were faithful to the book, and took everything one step at a time. Well, some of you might have skipped the introduction for which I'm sorry. We did mention there that the coding and testing cycles went hand in hand. They do occur simultaneously. In the last chapter we discussed how you could be testing your program as it was being written by adhering to some rather simple guidelines regarding how the program was coded. The same principle applies here. There are several stages of testing. We can call them "unit testing," "integration testing," and "system testing." These terms refer to the testing of the program, the part of the system the program interface with, and the entire system. If, for example, your program is the only program in a system (a stand-alone program) then the integration testing may be skipped. The unit testing is done to make sure that the individual

programs work according to their design specifications, that is, that there are not apparent errors. Integration and system testing don't start until all of the units are properly working.

Since we've already mentioned a "test plan" let's take a look at what goes into such a plan. Obviously we must define what it is we are testing, what the input will be as well as what the expected output will be. This should be done for each and every test that will be performed. Using this as a guideline you can begin performing the tests. Unfortunately we who use microcomputers must usually violate one of the first principles of testing — we must test our own programs. The reason that this is not recommended is that most programmers would rather prove that their programs *work* rather than trying to make them fail. Again, we normally test from the wrong basis, trying to prove that there aren't any bugs rather than trying to find bugs.

Since we can usually find what it is we are looking for, there are two things to keep in mind. One relates to the results of the testing; we must always *carefully* examine the results to make sure they are what they're expected to be. How often have we looked at something and seen what we expected to see rather than what was really there? The second point is that if we are looking *for* bugs we have a greater chance of finding them than if we are looking to prove that there are *no* bugs!

Another principle of testing is that when the "test cases" (the data/processing being tested) are planned they should include data that is reasonable and good as well as invalid and unexpected data.

Remember, most programs are subjected to more testing in the first hour that a user has it than it will be in most cases during the *entire* test cycle of the producer! If your testing is better refined (and defined) you have a greater chance of a program with few bugs being released. Remember, rule 1 says that there's always one more bug!

Let me give you an example of unexpected data. If we remember the AREA calculation program you will recall that we have some specific tests, namely for a square, zero length or width, and an end condition. What happens if a *negative* number is input to the program? Whoa! Everybody *knows* that an area must be composed of positive numbers. Ok, granted that *everybody* knows that; will everybody remember that? What if we get someone that is a pathological program bomber and he *deliberately* inputs a negative number? Tough, you say? No, your program should be as "bomb-proof"

as you can make it, even against users like that. By the way, most users are like that from time to time. It may not be intentional, but it does seem to work out like that. I used to have a person working for me that could break *any* program. We'd do all our testing and, once we were sure it was good, give it to her to see where we'd failed in testing. She always found something.

That leads to the next philosophical point behind testing. We must test every program to see that it does what it is supposed to do. We must also test to see that it does *not* do what it is *not* supposed to do.

"Just what the heck does that mean?"

Funny you should ask, I was just about to tell you. Let's consider the U.S. Government and the Social Security Administration. They do most of their work using computers. From time to time they write "benefit" checks to those entitled to receive them. Now, if the amounts are correct and they are correctly posted we can say that these

programs are working correctly. Right? What if the programs are writing checks to people that don't exist? Then the program is working incorrectly. Of course, there's usually a human error behind this particular occurrence, but that's not the point here, it's possible that there's a bug in the program that causes it to generate a check for Quincy P. Ripoff every 1000 checks. If that's the case the program is most assuredly not right (even if Quincy himself told the computer to do that!).

There is a certain perversity of nature that is often called the Poisson distribution. This is a definition of a kind of random number. For example, the Poisson distribution can be used to predict the number of raisins in a square inch of raisin bread, the number of typos on a page of a book and the number of errors in a section of code. This is a rather round about way of getting to the next axiom of testing. There is an increased probability of find a bug in a section of code where a bug has already been found. What that means to us is that once we find a bug in a routine, we should test that routine again, there's probably another bug hiding in there. Sometimes the removal of one bug will bring others to light. In any case, there seems to be a good bet that there will be more bugs found there.

One final idea while we're talking about the philosophy of testing. A test should never be planned or designed under the assumption that no errors will be found. Remember, a successful test is one that *finds* a previously unknown bug. Again and again I will be repeating that idea, testing is to find bugs, and is not primarily to prove that a program works! As long as that idea remains foremost in your mind during testing you will have little difficulty in deciding just exactly what you are to do when testing a program.

Alright, enough with the theory. How do you go about applying these neat theories? Actually that's easier to do than to write about. As the program is being designed there is an understanding of what the program is to do. At that point it is possible to define the results of the various levels of testing that will be taking place. For example, if there is a routine to calculate certain discounts based upon price and quantity it is a simple matter to define some test data, both valid and invalid, that can be fed to that routine. With that data and an understanding of what the routine is supposed to do you can write a test plan. When that test plan is actually executed, the output can be evaluated. If the results are exactly what was expected the test criteria is met. If, however, there is extra output or the program fails

to produce the expected results the test was successful, the program failed indicating the presence of a bug!

See how backward testing is? A test succeeds if the program fails, and the test criteria are met if the program doesn't fail. Seems strange and probably feels wrong in a way that's hard to describe. Don't worry about it if it does. Lots of people who make their living in the programming field still test the wrong way because they can't bring themselves to look at the possibility that their code might be wrong. As a result they try to prove that their code works instead of looking for bugs.

Testing a program requires a controlled environment. By that we mean the program under test should be isolated from any changes made either to the computer or the data coming into the test. Make sure that you are testing the program and not something else. If the program is being changed, it makes sense to change one thing at a time. In other words, control what it is that is being done. If you change the program and the hardware, how will you be able to determine which it was that affected the outcome of a test?

Another approach to testing is one that lends itself to the professional environment almost as well as it lends itself to the microcomputer world. This is called the "walk through." There are some advantages to this technique as well as some very definite disadvantages. We'll talk about the disadvantages first because they are such as to either make the technique work or fail miserably.

Are you often proud of your code? Will you become defensive if someone shows you another (not necessarily better) way of doing something? If you answered yes to *both* of these questions there may be some difficulty for you to utilize the walk through technique! The reason for saying that is based upon the necessity of your being able to explain the inner workings of your code almost as if someone else had written it. The term that is used for this is "egoless programming." What this means is that you can evaluate your own code openly, honestly, and without the need to defend the code when it is questioned. Sound easy? It isn't!

Actually, walk throughs are used at several stages in the life of a "professional" program; the design cycle, the test plan design cycle, the coding cycle and the testing cycle. Because there seems little reason to use that logic for a microcomputer, we'll simply use it as part of the testing cycle. There's some question, however, regarding whether this is properly placed in "testing" or is better

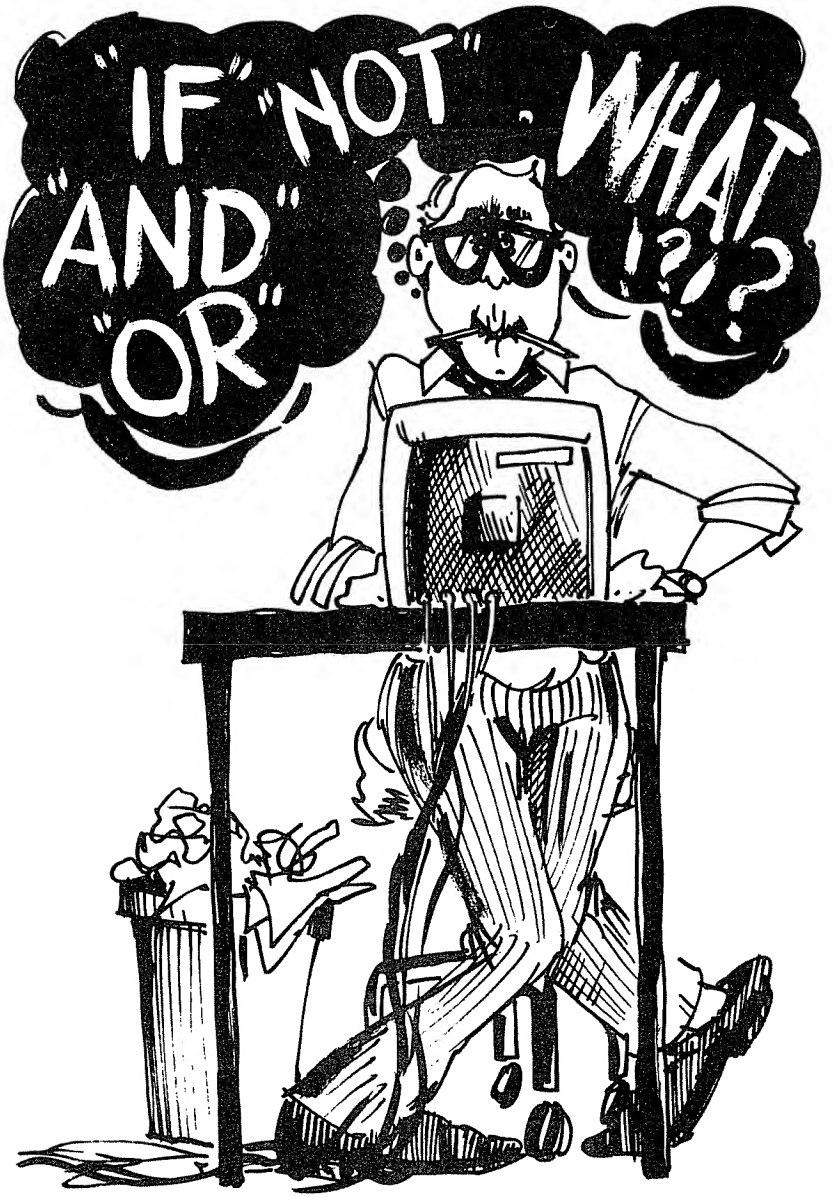
called a part of "debugging." There's a place for it in each phase. In the testing cycle it can be used to ensure that the code has followed the design. It merely requires going over the code (not necessarily line by line) and showing how it meets the requirements of the design.

Why would I want to subject myself to this "humiliation?" Actually, if that's the approach that will be taken, don't bother, it won't work! If however it can be approached with a sense of challenge, with a desire to let your peers evaluate your work in the cold, hard light of day. This can be a time when some will say how neat your code is and others will say "why didn't you do it this way" without regard for the personal touches you put in the code. Don't feel bad, that happens to even the best of programmers!

Actually, as the people performing this kind of review get more and more experience at doing them, they'll spend more time on getting to the real reason for the review. Questions will be more pertinent. Questions like "How does this work?" and "Where's this function?" will replace the "It'll run faster if..." sort of statement. The end result will be that *everyone* will learn. Your techniques will be picked up by those reviewing the program and you will pick up their ideas and techniques. It can be quite beneficial.

Some of the things that can be looked for in a walk through include looking for references to variables that haven't been initialized (a big source of erroneous results), making sure that references to tables (subscripts) are within the DIMension boundaries and making sure that uses of intrinsic functions (STRING$, MID$, ATN, etc.) all have the correct sort of parameter (thus avoiding the TYPE MISMATCH error).

Another source of errors (especially with structured programming) is making sure that comparisons (IFs) that are nested perform



the proper compares. IFs have some other major problems; they can be confusing when used as compound or complex statements. For example, to perform an action based upon one of two conditions

where A must be equal to or greater than B *and* B must not equal C *or* A must be less than B and B must be equal to C would be written as:

```
IF  (A>=B AND B< >C) OR (A<B AND
    B=C) THEN ...
```

But if we wanted EXACTLY the reverse of that condition we must remember that OR becomes AND and AND becomes OR. Thus the compare becomes:

```
IF (A<B OR B=C) AND (A>=B OR B< >C)
THEN ...
```

Is this correct? Let's re-expand the statement into English and see. We said that we will perform the action if A is less than B and either A>B or B< >C *or* if B=C and either A>=B or B< >C. Let's get rid of the obvious impossibility and see if we've still got the same thing. Again, the task will be performed if A is less than B and B is not equal to C *or* if B is equal to C and A is greater than or equal to B. See, it's not the same thing. In reversing the compare we goofed! It should have read:

```
IF (A<B OR B< >C) AND (A<=B OR B=C)
THEN ...
```

Where we can now say we'll perform the action if A is less than B and B=C *or* if B< >C and A>=B (eliminating the impossible condition of (A>B and A<=B) and (B< >C and B=C). The statement is the same, but much harder to read. Use the easier form (if possible) but be careful if you must turn it around to keep the logic consistent!

The other problem with the IF statement refers to nesting of IFs and making sure that there are the proper number of ELSE...THENs for the logical implementation. That means that there must be a corresponding ELSE for every IF except, possibly, the last one. More problems have arisen over the improper use of nested IFs than just about any other command (except GOTO). As usual, exercising the proper amount of caution during the coding cycle will usually prevent the problem from occuring at all. There should be no difficulty finding this sort of problem during testing.

Another major area of problems which we pointed out in the last two sections when we talked about writing the program is the module interface. This is not as big a problem in BASIC as it is in PASCAL, FORTRAN, COBOL or assembly language. The major source of problems in BASIC will occur in USR calls or in the use of the CALL statement available in Microsoft's BASIC-80. The problem has to do with the number and type of the parameters. This problem may also occur when using built-in functions. For example, you can't pass negative numbers to the SQR or LOG routines. This area needs to be examined carefully to ensure that there is sufficient coding to prevent this occurance.

File handling can lead to another source of errors. Here there are so many kinds of errors that we'll spend a little time exploring the sorts of errors that can occur. Most of the common BASIC interpreters today support both sequential and random files. It is important that the programmer make sure that the program is processing the correct form of a file. If you open a random file for output as a sequential file you will probably destroy the usefulness of the file for programs that expect to read the file as a random file (which requires that all records be the same length).

Making sure the file exists before it is opened for input or that there is sufficient error handling to catch the fact that the file is not present is important. Some computers (especially the Apple) will simply create the file if it doesn't exist and then the error will be transfered to the first read where there will be an end-of-data condition set. This again must be tested for.

Improper FIELDing of data to be placed in random access files can cause severe problems. This is an area where the length of a field is important. Another problem that is related to this is the improper use of LSET and RSET (for those computers that support them). These are used for STRING variables; MKI$, MKS$, and MKD$ are used for putting numbers in the file while CVI, CVS, and CVD are used to retrieve them from the file.

Random files seem to cause a lot of problems because of the way in which the "buffer" or temporary storage area is used. One of the more common symptoms of this problem is the "unexplained" occurance of the same record many times throughout the file (apparently at random). This is usually caused by updating a record without first reading the record. Another cause is getting a record for update and writing it back to a different location in the file by

PAY ATTENTION TO YOUR I/O AND DON'T BE DISTRACTED !!

changing the relative record number. In either case, the *entire* buffer (usually a sector long) is rewritten and all of the data in that sector will be written *wherever* you tell it to go. This can result in the other

records in the buffer being written along with the desired record while the old records in the buffer are overlaid by these new ones.

I once helped a friend track down a serious problem. He was opening a file for OUTPUT and then closing it again very quickly without doing anything. This was to force the file to exist if it didn't already. That seemed to make sense. The problem that he had was that, although the routine worked fine if the file didn't exist, when the file did exist it was *always* treated as if it were *empty*! Of course it was. Once the file had been opened for output it was treated as if all of the existing data was "throw-away" data and the end-of-file pointer was reset to the beginning of the file.

Another serious file related problem is one that can creep into BASIC very easily. If you have a file that contains five variables in a record, three numeric and two that contain strings, there is the potential for problems. If LINE INPUT is used to read the data from the keyboard and there is an imbedded comma it will process beautifully. You can write it to disk without any problem. Try and read it back using INPUT. The comma is, of course, a field delimiter. This will cause an extra field to be placed in the file which will throw all of the other data off by one field. Because BASIC does not treat sequential records as records but rather as a collection of fields there can be some nasty problems involved in removing bugs of that nature.

While we're on that subject, let's look at another easy place for problems to occur. It is sometimes advantageous to write a record at one time, and then to read the data back in parts. For example, we'll take our 5-field record again and this time we'll read back the first three fields in one part of the program and the other two fields in two different locations. If there exists the possibility that one or both of these other fields might not be read in, you can bet that it will happen. This will again lead to a file that is out of synchronization with the program. This is another problem that can be very hard to track down. It's especially so if either all of the fields are string or all numeric. Now there won't be a TYPE MISMATCH error to indicate the problem, the numbers might just be wrong or an address might get messed up, or whatever.

Another testing procedure is called "desk checking" where you will sit down and read the code you have written to see if it will work. This can be (and usually is) relatively unproductive. This approach is such that only the most obvious of errors will be caught. The more typical errors are syntax errors, not logic errors. By the way, that

points out an important fact. When testing, there are actually two different kinds of errors. The first and simplest kind is the syntax error, typing the keyword or command wrong, using a dollar sign when it isn't needed, forgetting to use parenthesis for a subscripted variable, and so on.

The second type of error is the logic error. This is the sort of error that we are really looking for. Desk checking can be useful for some logic errors such as a GOTO that lands at itself because it was left over from the testing stage. Other errors that are common are line numbers that are left out because, at the time, the line number was unknown and you failed to go back and fill it in. For this reason desk cehcking can be fairly useful, and is recommended before the first full-scale test of the program. Obviously if you have been using the "test as you go" approach the effectiveness of desk checking is greatly reduced because you will have already removed most of these common, obvious errors.

While you are testing there are actually two cycles going on, testing and debugging. These two operations work hand in hand to effect the production of a running program. In the next chapter we'll be seeing how we can use the testing techniques to simplify debugging. We'll also see how debugging can be done in an (almost) painless fashion.

There are, of course, lots of different ways to test programs just as there are lots of different ways to debug programs. In this chapter we've tried to look at the more common areas that problems like to hide in in order to determine how to test a program. In the next we talk about how to get those bugs out into the open where we can stomp them out!

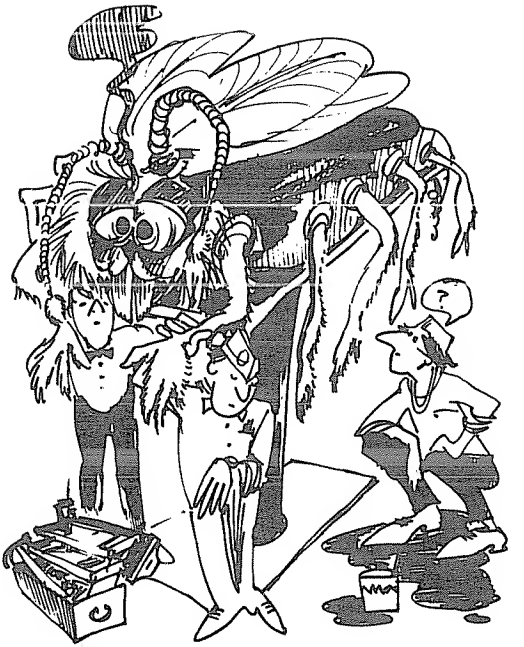
CHAPTER 11

Fixing Bugs the Easy Way

Having arrived at this stage, we are now in trouble. If we have a bug we have a problem, ergo, we're in trouble! No, I'm not Sheerluck Bones, that's just a statement of fact. Once we've found that we have a bug we have two major problems; we must identify the cause of the bug and then we must fix it. In this chapter we'll be addressing some easy approaches to those two tasks. We'll also take a look at some common causes for bugs and some ways to prevent them. All in all, this is going to be a fun chapter!

Before we jump into all of the goodies, let's stop and make sure we understand what a bug is. After all, how can we get rid of 'em if we don't know what they are? A bug is defined as a defect or error in a coded program. Thus it is a malfunction or a mistake. What that means to us is that we've done one of any number of different things. We've either told the computer to do something it can't, we've told it something other than what we intended to tell it, or we've failed to tell it something we did intend to tell it.

Obviously then there are some simple solutions. If we told it to do something it can't, we "un-tell" it. If we told it something other than what we thought we told it we correct *our* error and tell it correctly. If we failed to tell it to do something we just add the code to cause it to do whatever it is we originally wanted. Sound simple? It may or may not be, but the perversity of nature says that it won't be!



One difficulty in making simplifications like those above is that the real world is often not simple. For example, let's assume that we told it to do *everything* we wanted it to do, the computer could do everything we asked it to do, and we asked it in a fashion that it could understand. Now, still assuming all of that, let's also say that the program doesn't work right. Why?

Again there are several possible reasons. Maybe we've told it to do things in the wrong sequence. Maybe we've never executed all of the statements in the program because of some logic errors. Maybe we were wrong in our first assumption! That's three "maybes" that we've got to deal with. Actually, if debugging were as simple as it seems at first glance, there wouldn't be a need for this chapter!

First some simple guidelines. When the program first starts its test cycle, make sure that any error handling (ON ERROR GOTO) is either *very* carefully controlled or not used at all. The reason for this is that an error condition might crop up and be intercepted by an *error* trapping routine designed for some other error. For example, you might set an error trap to catch a non-existent file before opening it for INPUT, and then fail to reset the error trap. Thus, when a divide by zero error occurs later on, the trap for a missing file is entered and you've got no idea how or why (other than by looking at ERL on the TRS-80 and other computers that use BASIC-80 to see where the error occurred).

Now let's take a hypothetical error and see how we would approach it. The error could be anything, and the code could be anywhere in the program. What we're going to be doing is looking at a general principle that can be applied *anytime* there's an error that needs to be fixed.

An error has occurred and the line number is displayed on the screen. The program has terminated and the system is now waiting for us to do *something*. On the TRS-80 we may or may not be in EDIT mode. Whatever you do DO NOT CHANGE THE PROGRAM YET! Any changes will eliminate the tracks that are available for us to look at, including data values.

Our first step is to list the line we are currently processing. Is the statement syntactically correct? If it is, are the values being processed reasonable? If you don't know, list 'em on the screen, use the PRINT command in immediate mode to look at them. Make sure the numbers are valid for any intrinsic functions that might be being used. Evaluate the type of error. Maybe it will tell you what has happened and maybe it won't. For example, the message "ILLEGAL FUNCTION CALL" is nearly useless since it could mean *anything*!

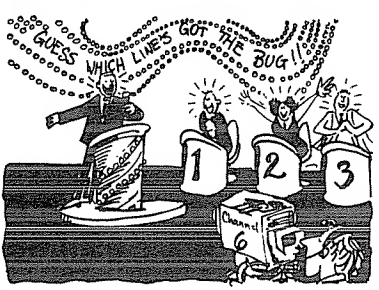
Our next problem is to determine if we have the line that *caused* the problem or merely the one that displayed the *symptom* of the problem. It's possible that we've gotten a divide by zero error and it seems obvious that the denominator is correctly zero therefore we need a trap to catch divide by zero errors at this point in the program. That's the *worst* assumption that could be made right now! The

divide by zero may only be symptomatic of a problem in an earlier statement where the dividend is calculated and one of the variables has not been initialized and is still zero. Multiplication by zero generally yields zero! Remember, ask yourself is it the *cause* or the *symptom* that you're fixing.

Actually the example above is relatively simple because the program has been terminated as a result of a condition that the computer didn't like (it couldn't do it). What about those times where

the test case was successful, that is, a bug was detected by virtue of the fact that the expected results were not produced? That can be a little harder to find. There are some common approaches to this and some, unfortunately, good approaches that are not common. We'll take a look at both of these.

The first approach is the "guessing game" approach. That's where the programmer says, "Because of the kind of error it *must* have occurred *here*!" Having done that he uses the trace facility (TRACE) to keep track of where the program is going. The difficulty with this approach is that the line numbers go by awfully fast making it difficult to follow along in the listing. The second problem is that the amount of data produced is usually quite large. A third problem



is that this only looks at the logic flow of the program and the problem may be *data* dependent.

Ok, let's add some PRINT statements to display the data as it changes or as we pass certain points in the program. This is fine, and again produces mountains of data that need to be evaluated. But did you display *all* of the pertinent data or was some omitted? Also, if this data is displayed on the screen there's a limit to how much data can be retained, and therefore it is of limited usefulness. Of course, hard-copy versions are better but it's still difficult to wade through all of this material. A final drawback is that it allows the machine to attempt the debugging while the programmer becomes merely a spectator. Debugging, like sex, is not a spectator sport.

This approach to debugging has often been called the "brute force" method. It has some advantages in that you can see *exactly* where the program is going, but it does eliminate the personal touch.

After all you should be using your brain to decide what the bug is, not letting the computer tell you. Look at it this way, if the computer knew where the bug was and what it was, then the computer could *fix* the bug. Obviously that's not going to happen!

What we want to stress is the fact that the programmer (you) should think your way out of a bug. There are a number of approaches; we will be addressing two. These two methods are relatively simple to describe and, in fact, you've already been exposed to them in school. For want of better names they can be called "inductive" and "deductive" approaches to debugging.

Each of these methods requires that you be familiar with the program. Therefore, we'll begin by assuming that you are debugging your own program. If you are debugging someone else's program you must have available the documentation for the program. This

should tell you what the program is supposed to do along with how it does whatever it is that it does.

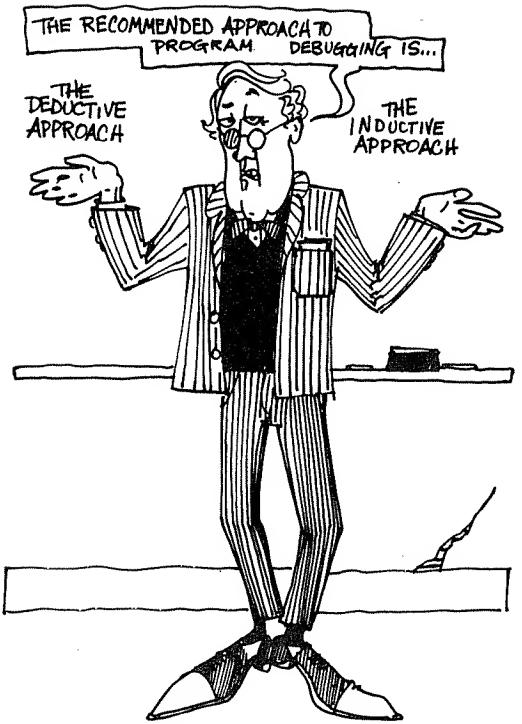
The two basic approaches mentioned above require that you take two different approaches. The "deductive" approach requires that you begin by examining the error and attempting to define areas of the program that *could* have been responsible for the problem. The "inductive" approach requires just the opposite, begin with the data that indicates the presence *and* absence of an error and work from that toward the problem spot in the program. We'll be spending a great deal of time in covering these two methods.

We'll begin by saying that all debugging processes can be carried out *without* a computer! Remember, I said "can" not "are." What that means is that the programmer can utilize his own knowledge of a program to arrive at the problem area without resorting to the use of the computer *if* he will follow some simple principles. These principles are all good, common sense ideas that have probably been mentioned by almost everyone in the data processing industry over the last ten or fifteen years. They've also been overlooked when they've been needed by all but a handful of programmers. Just what are these mysterious principles? How can we apply them to our own needs with a microcomputer? Does it cost extra?

Well, I'll tell ya, with tax, license and dealer prep it comes to $0.00 extra! That's right, it's absolutely free! Now I know that we've all been told that there's no such thing as a free lunch, but this isn't lunch — it's getting rid of bugs. It's sort of like the difference between using a home remedy to get rid of nasty crawling things from your home as opposed to calling in a professional exterminator. Both will work (assuming your home remedy works) but the former is much, much less expensive.

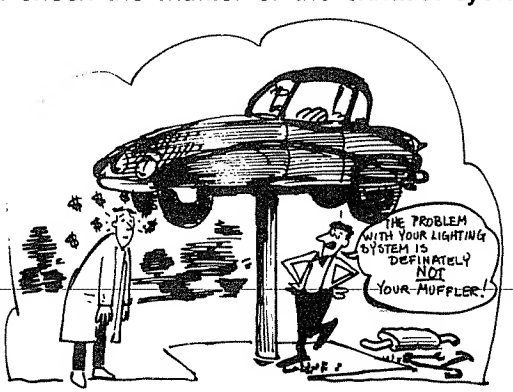
However, since there's no such thing as a free lunch, there is a minor cost involved — it takes being willing to *think* about the problem. If you've ever watched any of the game programs on television (whatever that is) and tried to figure out the hidden message, or puzzle or whatever, you've already applied some of the techniques that we'll be using in our debugging. The major asset here is an ability to think logically to a conclusion.

We'll begin by discussing deductive debugging because we are more apt to be familiar with these techniques. Deductive debugging requires that we begin by taking the areas of a program that might *possibly* be responsible for an error and then attempting to deduce



which one really is responsible based upon being able to eliminate one or more due to clues or data that might be present in the test case output, or in the program logic itself.

This might be compared to the work that is done when you take your car to the shop for service. If the lights don't work the repairman will seldom check the muffler or the exhaust system — the first

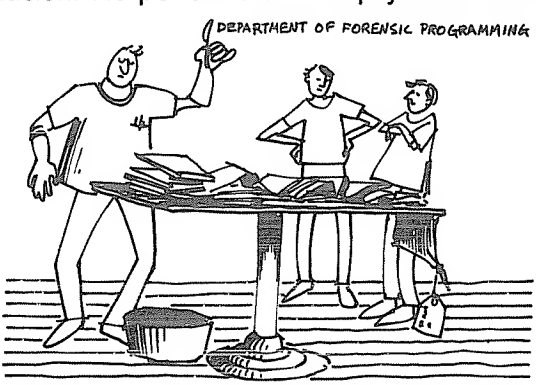


deduction indicates that the cause will be in the electrical system. If the radio and the horn works then the problem is probably *not* a dead battery or a blown master link. In fact, at this point you're probably already guessing that it might be a blown fuse or circuit breaker. See, deductive reasoning does work!

The same approach is used when taking a program apart. You can say that records that are out of sequence (when they're supposed to be in sequence) could be caused by a failure of the sort routine, by the fact that the sort routine wasn't used, or by some other routine that acts upon the data that is supposed to be sorted. If some data is sorted correctly and some is not, then it may not be the sort routine, or there may be some special conditions under which the sort routine fails. That is the sort of thinking that must go into finding a bug using the deductive approach.

So, to reiterate, the steps used with deductive debugging are to list the possible causes of the problem, eliminate as many as possible by examining the data thereby refining the original list of possible causes. When you are down to one or two remaining possible causes it is then up to you to prove conclusively that one or the other is the cause of the observed bug. This is done by actually "playing computer" and walking the known data through the various steps making the calculations as you go. This is, of course, a time consuming process but it may yield very good results. You can either prove or disprove the theory that you have found the area that is causing the error.

The next process, inductive debugging, is one that takes a little more work. It is more like what Quincy the Medical Examiner does on television. It is a piecing together of many small parts to arrive at the conclusion. He performs an autopsy and looks at the data



gathered as a result of his work (a programmer is often said to be performing a "post-mortem" when debugging). As the first step of his data analysis he organizes the data by category, carefully noting any discrepancies in the gathered data. Once this is done, he will formulate a hypothesis regarding the cause of death. If there is sufficient evidence (like, say, a quart of poison in the stomach) he may theorize that there was an attempt at suicide, or that the deceased didn't know what he was drinking. He will then attempt to prove that this was the cause of death. It may be incidental that the individual in question also had been shot 392 times.

If the observed facts (data) do not agree with the requirements of the hypothesis he will try another one until he finds one that fits. This is the reason for so many of the early Quincy shows, there was always a policeman ready to call a death homicide and Quincy would prove that it wasn't, or vice versa. This usually took the better part of an hour, but of course the writers usually knew how it would come out in the end. A programmer trying to remove a bug from a program often doesn't know what the clues are when they're found, and may not be able to relate them back to the source of the problem. In any event, he will dutifully note them and continue. Eventually there will be enough data to lead one to a conclusion (theory) about what happened.

Both of these techniques require a careful evaluation of the data produced by the program. It is this data that will allow you to determine if the program has failed (that is, the test case was successful). These data are the clues which will eventually lead you to the bug.

"Ok, what if I use these techniques and still can't find the bug?" you ask.

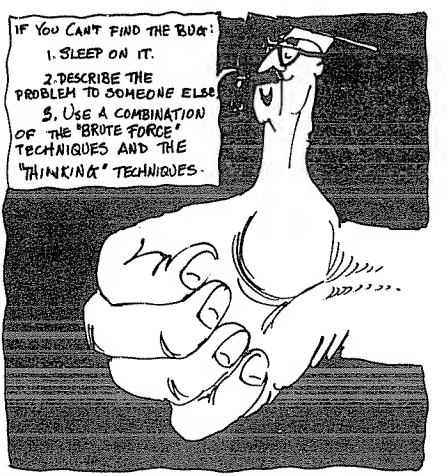
That's a fair question. More than once I've had programmers come to me and say they tried to follow the "logical" approach to debugging and no matter how hard they tried they couldn't find the bug! There are a couple of simple things that can be done immediately. First, don't beat your head against a stone wall trying to force the bug to disappear by sheer will power. It won't happen. Also, after about three hours you'll be so groggy and punchy from trying to find the bug that you might not recognize it even if you did find it.

What to do? Well, one time honored suggestion is to sleep on it. Let your subconscious mind work on the problem. It is usually better at that sort of thing because it doesn't care about such inhibitions as saying "it *can't* possibly be *there*." The subconscious also

works full time, that is, 24 hours a day.

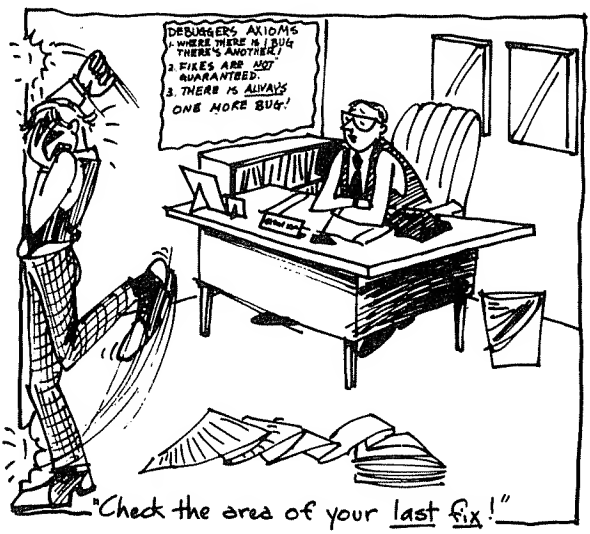
If that doesn't work, try describing the problem to someone else. I've found that telling someone else about the problem has brought the source of the problem out into the light simply by my explaining how a given routine "has" to work. Sometimes I won't see the error of my ways, but the person to whom I'm explaining it will. Seems he's not as close to the problem as I am and can evaluate it from a more level headed position than I can. Whatever the reason, this seems to work about 80% of the time. For the remaining 20% there are some other approaches that may be tried.

If you've tried all the "thinking" approaches and don't seem to be getting anywhere, then try to use an intelligent mix of thinking and "brute force" methods to arrive at the cause of the bug. I emphasize the word "combination" because you should never quit using the thinking approaches. The use of the brute force methods should be used as an aid to gather more data which may be used to further evaluate the problem.



Another reasonable method is to try running different sets of data through the program. Before you do that, however, make sure you have carefully decided what the results should be so that you can evaluate the results from the proper reference point. This is another way of getting additional data for the evaluation purpose. Remember, the more *useful* data you have the better your chances of finding the bug are.

One point should be mentioned. Assuming that you still haven't found the problem there is bound to be a certain amount of frustration setting in. At times like these it's usually nice to say "well, I don't know what's wrong so I'll try changing this and see what happens."



It doesn't matter what "this" is, it's still a bad idea. First of all, there's very little hope that the changed line is the cause of the problem. In fact, it's 1/n where "n" is the number of statements in the program (assuming that each line has an equal probability of being in error).

The other reason for refraining from this sort of activity is that it can result in masking the error, or causing other errors. Remember, any code used to make a change or remove a bug is more likely to be in error than the original lines of code were. These little axioms are all quite nice, but how then does one go about removing a bug that can't be found using any other ideas suggested so far?

Unfortunately, there really aren't any other ways of finding bugs. There's the "guessing" method which says that the most likely areas for containing the bugs should *all* be rewritten to make sure that the bug gets caught. The reason this is not recommended is that it is worse than just making changes at random. There are a couple of good reasons for this. First of all, if there was a bug in code that was carefully planned and laid out, then how much greater are the chances that there will be a bug in some code that is hastily conceived and implemented "just in case."

Assuming that we've found the bug, just how do we go about correcting it? That's not always as easy as it sounds. There's always a tendency to say that finding a bug is almost the same thing as fixing a bug. There's nothing further from the truth! Once a bug is found the work is really just beginning! Identifying a bug means that you may now begin evaluation of the error and its effects on the rest of the program.

Very few bugs exist by themseives. Most of them reside within a program and *do* affect the operation of the rest of the program. Before any changes are made it is important to evaluate the scope of the problem and to make sure that the fix will not introduce another bug somewhere. This is another reason that the documentation for a program must be fairly complete.

There are some principles that govern the way in which bug fixing should be approached. One, told to me by an old-timer, said that you should never fix bugs when the heat is watching. I'm not sure what he meant by that, but he seemed to know what he was talking about. Word had it he was one of the best fixers on the strip.

Seriously now, the first premise, which we've already mentioned, is that bugs don't like to live alone. Where there is one bug, there's probably more. It's almost a universal rule that if a programmer has let down his guard sufficiently to allow one bug to creep in he's let many in. This is due to the fact the each line of code that is written is related to the other lines of code around it. If there's an error, it may cause other errors by virtue of the way in which that error affects the rest of the program.

We've also discussed some other very important ideas. First of all, we must always try to fix the bug, not merely the symptom. Don't be confused and fix the thing that appears to be the bug without concern for what the real bug is. We pointed that out when talking about a zero divide. The error caused by dividing by zero may simply indicate a problem elsewhere in the program. You can determine this by evaluating the data — can the denominator *ever* assume a value of zero validly? If it can then you'd better be checking for that condition. If it can't then a check for a zero divide error will merely mask the real problem and will correct the symptom without ever touching the bug.

Another point that's already been covered is that the code being used for a fix is not guaranteed to be correct. This simple statement is often ignored by more programmers and, not surprisingly, is the

cause of more follow-up errors than anything else. What that means to you and me is that we need to be more careful in the application of fixes. Make sure we've got the real problem, and then evaluate it carefully. How best can we fix it? What will this fix do to other routines in the program? Is the fix compatible with the original design and intent of the program?

These are all questions that we should ask ourselves each and every time we work out the code for a fix. Additionally, we need to spend time *designing* the code that is used for a fix. Just as we carefully designed the program in the first place so too we must take the time to design the fix. This code is just as important as the original code (perhaps more so) and must be fit in so smoothly that the fix becomes invisible. That is, it must not affect the readability or usability of the program in a negative sense.

How's that again? What we're saying is that fixes often stand out like a bright red flower in a field of green grass. There's nothing worse than a fix that calls attention to itself.

"Why is that?"

Remember, part of the ideal toward which we are striving is a program that will be easy to read and easy to maintain. If a fix becomes a flashing neon light the next time the program is worked on the fix will get undue attention, perhaps to the detriment of the rest of the program. This can lead to the programmer giving the fix more priority than it deserves. All of this can lead to the idea that the program structure is not what it appears to be. We want to keep the program both visually and logically structured so that maintenance is easy to perform, but also so that the program works as efficiently as possible.

Now, with all of that in mind, let's look at a corollary of the original statement. If a fix for an error is not guaranteed to be correct, then the possibility of the fix being in error increases as the complexity of the program grows. What that means is that as the number of lines increases, the chances for a given fix being correct decreases. This again ties back to the idea of the relationship between the different lines of code.

What does this mean to us? It means that when we've finished installing the "fix" we need to go back to the test cycle with an awareness that the "fixed" area is now more prone than ever to fail. We should be especially careful to make sure that the fix is thoroughly exercised. In fact, some have even said that when the next error is found the first part of the program to look at is the area of the fixes, whether they can be pin pointed as possible error sources or not!

By the way, although this book is primarily aimed at the BASIC programmer, there is another important point that relates to the assembly language programmer. When removing bugs from assembled programs there is an almost overwhelming urge to make "patches" to the object code. Don't! The drawback to this is two fold. First there is a tendency to do this in a hurry which means that the evaluation and design of the fix is not done in a manner which is conducive to proper fixes. Additionally, it is entirely probable that the fixes will never be reflected in the source code so that the next time the source is assembled the bug is still there. Most unsanitary! Not only that, it makes for a great deal of frustration when you think back and decide that you've already fixed that bug once!

While we're talking about debugging, let's take one more topic and throw it out. Each and every one of us has a certain dislike for various forms of regimentation. As a result we like to be "individuals."

For programmers there is a very obvious way in which this is stated. We all have certain failings that are, unfortunately, somewhat common.

We don't all make the same mistakes, but each of us will have a tendency to make the same mistakes over and over again until we learn what the mistakes we are making are and find some way to prevent them. One such way is to keep track of the kinds of errors you find in your own programs. After doing this for a while you'll be able to formulate a statement of the kinds of errors that you've found and, hopefully, be able to define a way to prevent them in the future.

For example, if you know that 70% of your errors occur in nested IF statements you might spend more time when coding them to insure that they are coded correctly. That is, that the computer can do them, that they are what you want, and that they are stated in a fashion that is correct in terms of sequence and syntax. This will help eliminate errors down the road.

Now there is no such thing as a bug free program, just as there is no such thing as a perfect programmer. We are all going to fail from time to time (why, just last year I left out a PRINT statement that...oh, never mind). But as we carefully evaluate our failures and determine *why* we fail (rather than *how*) we might be able to apply ourselves to correcting the reason behind the failures. As with programs, we need to correct the bug, not the symptom.

Unfortunately there has been very little literature on this subject and so we're all working on our own. One thing of which we should all be certain is that there has never been a program that had no bugs (with the exception noted earlier), nor will such a program ever exist. Let me give you an example of a bug that was so severe that the fix actually *doubled* the size of the program. IBM (again!) has a program called IEFBR14 which is, in reality, just a way of getting access to the various system routines used in job initialization. This program consisted of *one* instruction which was a "BR 14" which means to branch (jump) to the address contained in register 14. Well, that seems ok, the system supplies the return address to a called routine in register 14 so jumping to it should be just like executing a Z-80 RET instruction. Actually, there was *nothing* wrong with the code in this program! The error lay in the what the system expected in another register when the program returned to it. Register 15 was supposed to contain a number that indicated the status of the job. This number should be in the range of zero to 4,096. In

the case of our little BR 14 program the number in register 15 was the ENTRY POINT of the program which was usually larger than 4,096! The fix was to add the instruction "SR 15,15" which subtracts register 15 from itself. This results in a return code of zero which was acceptable. The BR 14 instruction took two bytes and the SR 15,15 instruction took two more bytes thus doubling the size of the program!

The real example here is that sometimes the bug may appear outside of your program and yet still be caused by your program. Never make the assumption that *your* program is so simple it *must* be correct. After all, what could be more correct than a single instruction program? There's no such thing as a bug free program. That simple principle should probably be tatooed on the back of both hands of every programmer in existence.

This section has discussed both testing and debugging. While they've been treated in different chapters they do go hand-in-hand. They are related in that one is used to find what the other fixes. The title of this chapter perhaps gave you the wrong idea — that is that bugs were easy to fix — they aren't. If they were easy to fix the computer could do it for you! The idea here was the same as the ideas throughout this book; to present a methodology that would lead to a simpler way of finding and fixing bugs. One that will allow you to maximize your own personal productivity.

As strange as it may seem, finding bugs can be reduced to an almost science. All it takes is the patience of Job, the wisdom of Solomon, the steadfastness of the Sphinx, and the world's greatest sense of humor. Seriously, if you will approach each bug as a chance to prove to yourself how good you really are, if you can make it a game — you against the *bug* — there's a good chance that finding and *zapping* bugs will become a relatively painless process.

No, there's not even a sure-fire method of finding bugs. You've got to track 'em down through good, thorough testing practices and procedures. Remember to keep track of the ideas that you've had during the testing cycle relating to how you can improve your tests. This will allow you to build better test plans the next time around. After doing this several times you will find that test plans, testing and debugging will become almost like second nature. There won't be any more hair pulling (you'll already be bald), screaming (your vocal cords will be shot) or long, sleepless nights over bugs. Really, finding and correcting bugs can be fun!

Section 6 — The End of the Journey

We've arrived at the end of a long journey. We've described how you go about writing a program. We started by taking an idea and fleshing it out, adding material where it was needed to make a real, live program out of it. Now that we've got the program tested and debugged there's still one more step, one very important step. I'm sure most of you have seen the cartoon that say's "the job's not done 'till the paperwork's done."

This final section will cover the topic of paperwork — specifically the documentation that's required for a program. We've covered some of the material as we went along, but we'll cover it in more detail here so that you can get a good idea of what's really needed.

CHAPTER 12

Readable Documentation

The title of this Chapter says a lot about what we want to accomplish. A great deal of care is taken these days to make sure that a program is "user friendly." That's all well and good, but it must also be pointed out that a program is only part of the package. There's also documentation.

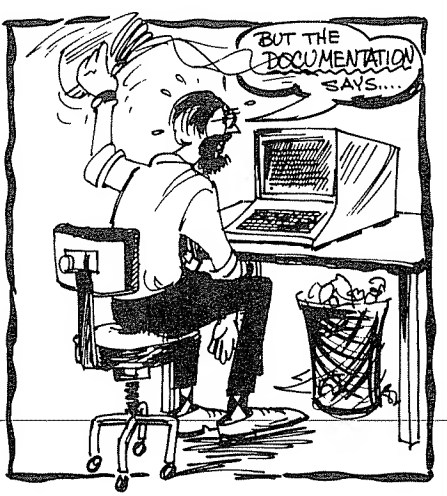
"But the program's just for me! Why do I need documentation?" you ask.

That's a fair question (you guys are sure on your toes with those questions). Why do you need documentation? We've already pointed out that you might be picking up a program that was written some time in the past. Let's imagine what might happen.

First we find a disk (or a tape) and we load the program that's on there just to see what it is. It looks neat and we decide to run it. Wait a minute, what do we need to run it? What does it do? Are there any special requirements before we start executing the program?

Without documentation we would have to answer those questions using the program itself as the documentation. Let's also assume that this program was written before you read this book and therefore the program is constructed in a rather haphazard fashion. Now

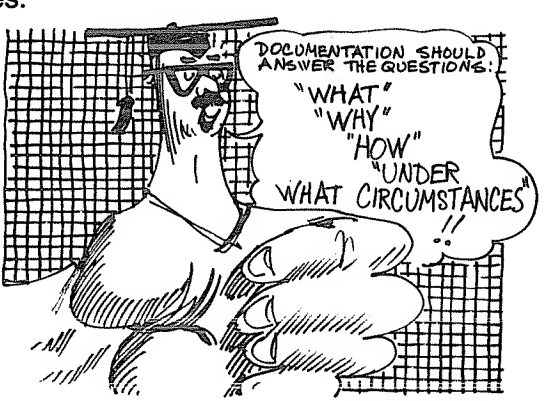
we have a real problem. It's been so long since we wrote the program that we'd forgotten about it and now we can't figure out what, if anything, it was supposed to do. In fact, we don't even remember if it ever ran. Oh well, it was a good idea anyway!



This is the sort of thing that we want to avoid. What we'll be covering in this Chapter is the way to document a program so that it is truly painless to do so, and so that we can decide how much documentation we need. After all, if the program is only for our own use there's no sense in writing a 500 page user's manual for a 200 line program!

Let's begin by setting down some simple guidelines about what should be in the documentation. We can then proceed to define how much should be used based upon whether the program is for your own use or is to be marketed. One important point should be made before we continue. If you are using this book to enhance your own skills and you work for a company in the data processing area, don't forget that each company has it's own set of rules and regulations regarding what should and should not be in the documentation. The material we'll be covering here has been generalized so that it can apply to a broad spectrum of the microcomputer users, not necessarily the commercial users. Therefore, caveat emptor (or be careful of who it is that you buy a used chariot from)!

If you ask thirty different data processing managers what should be included in documentation there's a good chance that you'll get fifty or more different answers. The most common answer is likely to be "It all depends!" And, believe it or not, that's perhaps the best answer that can be given. The question itself is so open, so nebulous that there's really no one answer that can satisfy it. What we'll be doing is to try and pinpoint the items that all of these hypothetical managers would have mentioned. These common points will probably address the questions of "what," "why," "how," and "under what circumstances."



Briefly we'll define each of these terms and then develop more complete answers as we go along. "What" is used to specify what it is that the program does. "How" therefore will tell us how the program does whatever it does. "Why" is the statement of the problem that the program was intended to solve while "under what circumstances" defines the operating environment under which the program will solve that problem. See how easy it is? Don't get over confident, if it were really that easy there wouldn't be several million programmers that hate documentation, or would there?

In addition to the points covered above regarding what the documentation should include, there are also several different kinds of documentation. By that we mean that you should have documentation regarding the program, user documentation, operator documentation and, possibly, control documentation. For microcomputer users the user and operator documentation are usually the same document, perhaps separated into different parts.

Let's begin by discussing the requirements for program documentation. That's perhaps the easiest to do since most of it has already been done! What we mean here is that the bulk of the program documentation will consist of the design documents as well as any changes made by you as you were coding the program. This will effectively answer the questions of why the program was written, what the program does and how the program does it. The amount of material here should be of sufficient detail that you can refer back to it at any time and be able to understand the reasons for the program, and the reasons behind the implementation. That is, that you can understand why you wrote the program the way you did.

In addition to these design documents there should also be a narrative for each routine which will describe the routine in general. This description should include any interface requirements such as variable usage, data returned, file status and so on. The narrative should also include a brief overview of the logic used in the routine especially for any code that is more complex than usual. This is primarily aimed at being a record that will allow later maintenance of the program. Of course, the "beauty" of the document is defined by the destination. If you're the only one that will ever use it (except for the possibility of a few close friends) then hand written in a reasonably legible form is probably acceptable.

Another area that should be included in the program documentation is examples of the files. Input, output and report examples

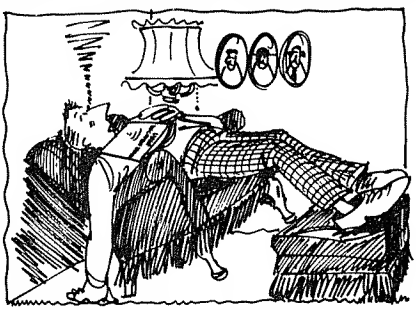
should all be carefully laid out so that you will be able to remember what the program expects in the way of data. The report formats are especially useful if you are trying to show someone else what the program will do. It's always easy to explain that sort of thing with a picture around. As usual, a picture can be worth a thousand words.

The user/operator documentation is designed to allow someone (other than yourself) to run the program. This requires a fairly complete description of the problem that the program is designed to solve along with a general description of how it solves the problem. The detail that is present in the internal programmer documentation is not only unnecessary but also unwanted. Most users don't care how a program does what it was purchased to do, all they care about is the fact that it does indeed perform as advertised.

The majority of the user documentation for microcomputer programs on the market today are awful! They assume that the users know everything that the programmer knew when the program was written. The documentation makes assumptions about the knowledge and skill level of the user and these assumptions are normally invalid! There's a principle that is always mentioned anytime anyone is talking about communicating an idea or an abstract concept to someone else — "KISS" or Keep It Simple Sweetheart. Some people have been known to change the last "S" to mean something else, but we'll use this more generous definition.

KISS. There's something inherently nice about simplifications like this. What it implies is that the material being written should make minimal assumptions about the reader; the terms should be those that the *reader* will be familiar with, the material should be presented in a logical, step-by-step fashion so that the reader can grasp the meaning easily.

We could be talking about several different things here so I'll begin by clarifying what I've just said. Whenever any material is written for someone else, the writer just naturally tends to use words that are familiar to *him*. This can be a severe limitation because the reader may not know the words. If you've ever watched children trying to read you're aware that they'll normally skip over words they don't know rather than taking the time to look them up. If there are lots of words that are unfamiliar then the child will tire of the book



quickly and go on to something else. The same idea holds true for the users of our programs. The documentation must be simple enough for them to read and understand.

Another problem that is found in documentation is that the reader just gets tired of reading after a while. To make this clear, how many people do you know that purchased computers that are now just sitting around gathering dust or are used only occasionally as a game machine? Why do you suppose people would purchase computers and then let them sit around doing nothing? The answer is very simple — the documentation makes it too difficult for them to learn how to use the computer! There are a great many people that can't learn simply by reading; they have to practice. This is sometimes called "manual learning" because they learn what they practice rather than what they read.

This is one of the reasons that homework is assigned in school. Many people can quickly grasp the concepts in class but must then practice to make the skill their own. Homework is not the best answer,

but it is the only sure way to provide practice. These principles must be considered when writing documentation. That means that there should be lots of examples. These examples are used to explain the basic operation of the program, from power-up to power-down if necessary! There should also be examples of just about every kind of action that the user can take with a program. Remember, Murphy has predicted that if anything can go wrong it will. This is usually interpreted as meaning that the documentation will omit some critical sequence for the novice operator which is likely to cause him to destroy the program, the operating system or the disk.

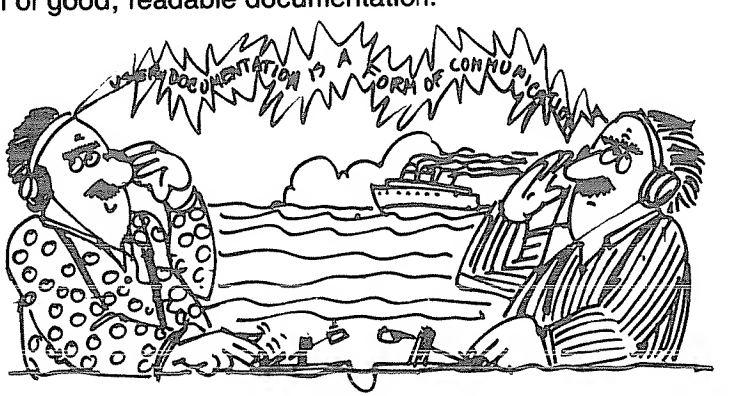
Another failing of most documentation is that it expects the user to know what data he needs *before* he executes the program. Often he won't know until he gets to the input routine in the program and suddenly discovers that he doesn't have the data and doesn't know how to get out of the routine he's currently in so he can go get the data. That's a most unfriendly situation to be in!

"Yes, a question from the back."

"Then, are you suggesting that the documentation must be as carefully planned as the program?"

Very good, that's exactly what we're suggesting. See, I knew that you were paying attention. As with everything related to the computer, the more time and attention put into the product the better it will be. This is true of any human endeavor but is especially so in the case of computers.

Now that we've talked a little about what has to go into the documentation, let's take a look at some of the mechanics of documenting. In other words, let's look at how you go about writing documentation. There's some simple rules that govern the construction of good, readable documentation.



The first idea that we need to keep in mind is that documentation is meant to communicate with someone else. The process of communicating is a very complex function that requires cooperation on both ends. The document that is being written for someone else to read has some rather broad requirements that are defined by the language that is used. Since we will normally be writing in English we must follow the rules of the English language. Since we are going to be communicating with people that are not necessarily "computerists," that is, people that are "into" computers, we must restrict the use of buzzwords and slang that is peculiar to computers.

If we address ourselves first and foremost to the *reader* we will be better able to document our programs. What does the reader expect from us? What is his background? Is he willing to work at understanding the documentation? These questions are really at the center of the problem relating to good computer documentation. By the way, this applies to hardware as well as software documentation.

By answering the questions we've just asked we will further define what must be in the documentation, at least from a mechanical standpoint. The reader expects us to tell him how to use the program, what data is required, and what the result of using the program will be. Further, he expects us to tell him this in terms that he is familiar with. He doesn't want to wade through a lot of technocratic double-talk. We will lose the reader if the terms used aren't common, everyday terms used by the average person. If the package is an accounting package aimed at the small business man it is permissible to use some accounting terms, but remember that the small businessman is probably *not* a CPA or an accountant so be careful of the terms used. At the same time, don't try and invent terms based upon common words that might confuse the issue. It is better to be correct and wordy than incorrect and terse. In other words, if you can't use the correct term because it might confuse, don't build a funny term just to eliminate using more words. Always take the time to explain the meaning of words if you must use technical terms!

When writing documentation, or any material for that matter, we must consider the audience that we are targeting our writing to. If we are writing a program that is used for calculating orbital mechanics we have a certain idea of the sort of person that might use the program and therefore an idea of the educational requirements.

There's still no reason to attempt to overwhelm our readers with a display of our prowess with the English language. They can expand their vocabulary with Reader's Digest at their leisure, let's not force them to do so unnecessarily with our documentation!

There are some other considerations while we're at it. The mechanics of writing require that we organize our thoughts into sentences and paragraphs rather than putting words on paper at random. To do this properly requires an understanding of how we read. Reading is a process of transferring data from our short term memory to long term memory in a series of images. These images are not related to words but rather to concepts. The human brain is an amazingly complex computer, and it functions in pictures rather than in words.

The human brain also likes to take advantage of "phrases" since the short term memory can only hold a few words at a time, usually around six or seven. This means that sentences that are long and complex may not be fully understood. For example, most of the sentences in this book are in the range of 15 to 20 words long. The longest is around 50 words and the shortest is one word. Because you are an interested reader you have made a sacrifice and worked a little harder at understaning some of the material that's in this book. You see, interest plays an important role in the gathering of data.

Since there's so much desire in American society for instant gratification there's a need to make sentences shorter and harder hitting. If you don't like a show on television you can always change the channel, if a program takes to long responding you hit BREAK just to see if the program is still running. This impatience is also visible with people that are reading. Try reading the Count of Monte Cristo or any other of the classics of that period and you'll find yourself skipping over long, tedious sentences that are merely used as "color" sentences. There's just no desire to read anything that doesn't immediately seem relevant.

If we understand that lack of tolerance on the part of our reader we can try and keep the material tightly written. This will eliminate some of the problems. So what we're talking about is writing that is easy to read and to the point. We want to minimize the number of words that are not in the working vocabulary of the reader. Are there any other ideas that we will need to keep in mind when we are writing our documentation?

We need to remember that, for the most part, our documentation is all that the user will see of our work. He will never meet us, and may never talk to us. It is therefore necessary that the documentation address as many of the points about the operation of our program as we can think of. We must also write the documentation so that the user will really believe that we care about him. That's part of what will bring him back to purchase from us again.

What if we aren't planning on marketing the program, do we still need that kind of documentation? Of course not, there's no reason that we would need that much for ourselves. On the other hand, we will want most of the same material so that we could pick up the program a year or two later and still be able to use it. We can take more liberties with the language since we are intimately familiar with the background of our intended reader. The use of buzzwords and poor spelling or grammar is acceptable here, but only marginally so.

The problem with doing anything on a marginal basis is that it can tend to become a habit, one that's tough to break. We must always strive to produce our best work regardless of whether it's for ourselves or for outside consumption. There's never an excuse for poorly written documentation that's incomplete or, worse yet, incorrect. Remember, even when writing for yourself you must realize that time can dull even the sharpest of memories, and you may not remember all that you think you will.

In order to make this whole discussion of documentation come full circle, let's take a look at how we might go about documenting the CHEKBOOK program. This will provide a reasonable approach at documentation since we have all developed the program throughout the course of the book and therefore should be reasonably familiar with the way in which the program works. For this reason, we won't write the program documentation. The book itself is more than adequate since it covers all of the design criteria as well as a narrative of each of the routines. All of the flowcharts covering the program are also included. This is, of course, more documentation than is normally needed. After all, who wants to write a book for each program? The documentation that we'll be writing, then, is the user documentation. This should clarify all of the issues that we've been discussing.

USER DOCUMENTATION FOR "CHEKBOOK"
An Automated Check Writing Program

The CHEKBOOK program was written to provide a simple method of making sure that all checks are carefully tracked and entered. Many times we have written a check and then forgotten to enter it in the register, or have failed to properly balance the checkbook. Using CHEKBOOK this problem will never happen. It will automatically keep track of the current balance for not just one but up to five different checking accounts.

This program is capable of writing checks with the assumption that you have an impact printer attached to your computer. This printer must be capable of handling your normal check stock, or you can order special continuous form checks from your bank. The program will automatically keep track of check numbers too!

To use this program you will need to have at least one disk drive (or a cassette recorder if you have the tape based system). Place the program disk in the first drive and "boot" the system according to the directions supplied by the computer manufacturer. This normally involves turning on the power or, perhaps, hitting the reset button. Enter BASIC following the manufacturers directions and run the program "CHEKBOOK/BAS."

The program will be asking for data related to your current checking account status such as the current balance, the next check number and so on, so please have this material ready when you run the program. The first time the program is run the Check Register File (called the CRF) will not be present on the disk, and there is no need for concern because the program will simply create this file for you.

After the program has gotten ready to begin processing it will display the "menu" or list of things that it can do. While it is getting ready it will display the message "SYSTEM INITIALIZATION IN PROGRESS, PLEASE WAIT" which means that the program is reading data from disk and doing some other "house keeping" to get ready to take care of your requests.

When the "menu" or list of options is displayed on the screen you will see a list of seven different functions that the program can perform. In the upper right hand corner of the screen (where the dashes are now) is the area reserved for error messages. The selection of an invalid code will result in an error message there.

Hit "ENTER" or "RETURN" on your keyboard to see the results.

We will briefly go over each of these functions so that you will know what they are used for. The POST routine is used if you have written a check yourself (a hand written check) and you want the system to know about it. This is a good idea so the balance maintained by the computer will agree with the one the bank has.

The WRITE routine is used whenever you want the computer to write a check for you. It will always use today's date and the current check number when the check is entered into the Check Register File (CRF). We'll talk more about the CRF in a moment or two.

The STATEMENT routine will print out statements showing all transactions (checks written, deposits made, etc.) for either a selected account or all of the accounts that the system knows about. This function will require the use of a printer so make sure it is ready before using STATEMENT.

Selection number four is the SETUP routine. This is used anytime you want to tell the computer about a new account. It is also used the first time you use routines 1 or 2 (POST and WRITE) if there are no accounts set up.

The DEPOSIT routine is used anytime you want to tell the computer that you have put more money into the checking account. The computer will add this amount to the balance as well as making an entry in the CRF.

Because there is a limit to the number of entries the computer can hold in it's memory, we've provided a PURGE routine as selection six on the menu. This will print a final check register (called the Month End Statement) and then will adjust the month starting balance. It will also get rid of all of the entries that have been made since the last PURGE or since the program was first run if there have never been any PURGEs. The program can hold up to 150 entries (deposits, checks, etc.) so we suggest that PURGE be run at month end, or on a regular cycle.

The END routine will cause the system to rewrite the Check Register File (CRF) and then the program will end. This makes sure that all activities that have taken place are saved so that the program will know what has happened in the past.

Each of these routines requires different information that must be supplied so we'll now take a quick look at what is needed. The POST and WRITE routines require some information in common,

both will want to know who the check is written to, the amount of the check and the account on which the check is drawn. You will be given an opportunity to supply a comment for entry in the CRF and, in the case of WRITE, on the check. In addition to this information, POST will want to know the date the check was written (if you hit ENTER the system will use the current date) and the check number (entering "-1" will cause the current check number to be used).

The STATEMENT routine will ask if it is to print statements for all accounts. You may answer with either YES or Y to indicate you want all accounts, or if you want to specify an account number you may reply NO or N. If you have elected to specify the account number the program will now ask for it. The statement will then be printed on the printer.

The SETUP routine will ask for the account number (you may either number accounts as 1, 2, 3, etc. or use the real account numbers). The starting balance and the next check number will also be needed by this routine.

The DEPOSIT routine will want only the account number that the deposit is to be posted to and the deposit amount.

PURGE will need the same information as the STATEMENT routine.

The END routine requires no information.

The information that is maintained by the program is all stored in the Check Register File also called the CRF. This is a sequential file stored either on disk or on tape. The file contains the account numbers, current balances and the next check number for each of the accounts that are setup. It will also contain a record for each check that has been written of which the system is aware as well as all deposits that have been entered into the system. The information is stored as a single group of letters and numbers with a delimiter that tells the program how to break this group apart.

The error messages that are issued by the program include the following. In each case we have attempted to indicate what action should be taken to correct the error. In the event that you can not recover from an error, you may usually rerun the program. As long as the END routine has not been used the data in the CRF file is still untouched, and you have effectively backed-out or reversed everything you have done.

1) ERROR, CHECK REGISTER FILE NOT FOUND. FILE WILL BE CREATED!

This message indicates that there is no CRF on the disk, and the program will treat this as the first execution of the program. After the END routine has been used once there will be a CRF and the message will not occur again. If there should be a CRF file check to see if you have the correct disk in the drive.

2) NO ACCOUNTS SET-UP YET.

This message indicates that the SETUP routine has not yet created an account that can be used. If this message appears after the selection of the POST or WRITE routine you will continue in that routine after establishing an account. If this message appears in the SETUP routine itself it is simply an informational message.

3) ERROR, ACCOUNT NUMBER NOT FOUND. RE-ENTER!

This message is used when the system can't find the account number supplied for the POST, WRITE, DEPOSIT, PURGE, or STATEMENT routine. Correct the account number by re-entering it.

4) BALANCE WILL GO NEGATIVE, WRITE CHECK ANYWAY?

This message is used when the WRITE routine detects that the amount of the check exceeds the balance in the account. You may still write the check.

5) ERROR, CHECK REGISTER IS FULL. HIT ENTER TO CONTINUE.

This error indicates that there are currently 150 entries for the account number being processed. Hitting ENTER will cause the computer to return to the menu and no action will be taken. This means that whatever you were attempting to do (POST, WRITE or DEPOSIT) did not take place, and the computer has ignored the request. This may be corrected by running a PURGE.

6) INVALID OPTION

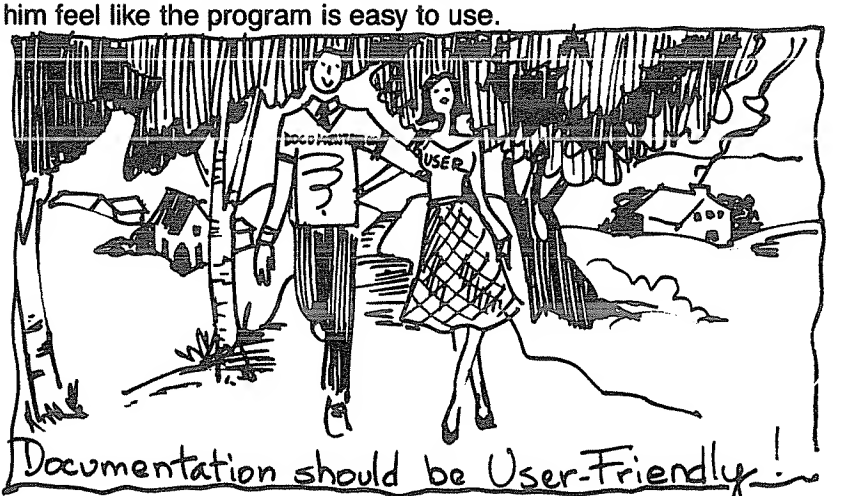
This message will only appear in the upper right hand corner of the screen when the menu is displayed. It indicates that you entered a number or letter other than the numbers 1 through 7 which are the only valid routine numbers. Please correct the entry.

You will note that the MENU routine does not require you to hit ENTER after entering the routine number. All other data must be followed by the ENTER key to register the data to the computer.

This concludes the documentation for the CHEKBOOK program.

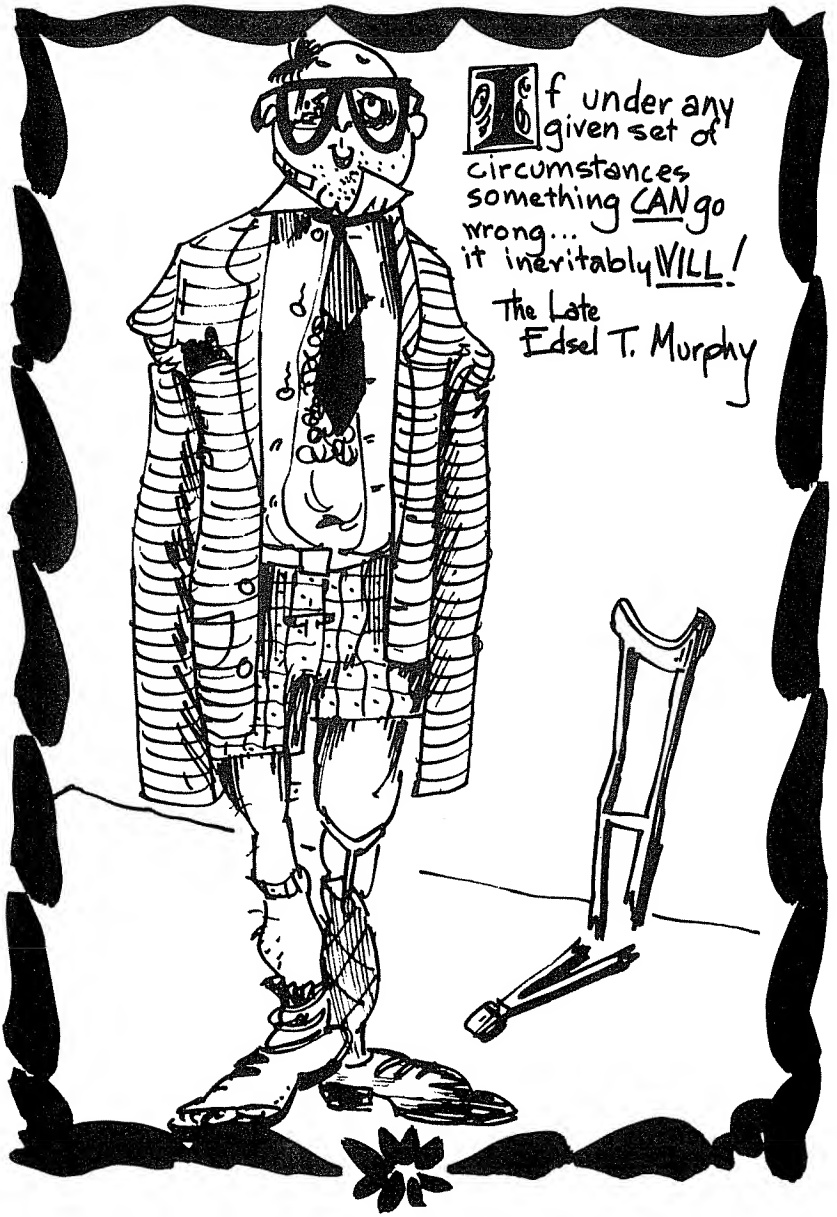
There it is. It's not terribly long, nor is it terribly complex. The documentation tells the user exactly what is needed to run the pro-

gram without burdening him with extra words. Of course, there's a lot more we could have said, but that's probably more than we needed to say. The idea is to provide a guide that he can follow and understand. If he can quickly start using the program we've probably allowed him to learn what the program expects, and we've made him feel like the program is easy to use.



The name of the game is to be user friendly, both in the program and in the documentation. We must always remember both sides of the task. If we forget one or the other we may lose a customer. If we are using the program for our own use we may later find that we have a salable product, but it's too much trouble to clean up the program and documentation to make it sell well.

As a final recommendation, always treat every program as if it were intended for the commercial market place. There may be times that that becomes a real pain in the neck (or elsewhere) but it will eventually become a habit that will yield a large number of benefits to you. You'll find that your programs are all much better (and each gets better than the last). You'll find that you are developing programs just as quickly and with fewer bugs. Who knows, you may even be having more fun!


If under any given set of circumstances something CAN go wrong... it inevitably WILL!
The Late Edsel T. Murphy

CHAPTER 13

Epilogue

Well, this is it. We've come a long way since we started this book. There's been a lot of ground covered and we've done some fun things (at least I did). I certainly hope that you've had as much fun as I have. There's a lot to say for developing a program the right way. There's always the sense of accomplishment that goes along with it. Especially if the program is relatively clean and requires minimal debugging!

Back near the beginning of this book (some 52,000 words ago) we mentioned that there was a principle of teaching that states that you tell the student what you're going to teach, then you teach the material, and then you tell 'em what it is that you taught 'em! We're now at that point in the book.

We've covered some of the more important of the program development techniques when we discussed the modular and structured methodologies. These two techniques will allow you to develop programs based upon work that you've already done. It's sometimes very frustrating to try and take material from another program when it's scattered all over the place. This is especially true if there's some code there that you really need to finish off another project. You've got to go through the program and evaluate the function that you want, determine which code is essential to the function and which is not, and then extract that code and make it useable in the new program. It's no wonder that so many programmers would prefer to rewrite and reinvent the routine than to try and extract it.

Perhaps of equal importance, we've finally laid down some guidelines that help us in starting a program. So many times we've begun our programming in the middle, completely skipping the beginning stages. And, with that handicap completely ignored, we've asked ourselves why we have so many problems. I ask you now, is it any wonder that the programs work at all? If they do, it's only due to your own dedication and tireless efforts.

Sometimes programmers will come to me and tell me that they have thought about becoming involved in data processing as a profession, but gave up the idea when they realized how hard it was — they didn't want to spend their whole lives debugging programs.

Well, let me tell you, there's no job easier than data processing if it's correctly approached! The money's good and the challenge is always there. The hard part of the job is staying current. For example, in the three months it's taken me to write this book there have been over twenty new computers announced in the microcomputing industry, there've been at least 10 new hard disk drives, and countless operating systems for the TRS-80.

Change is essential to life, and it's the driving force behind data processing in all of it's phases. We in the microcomputing industry are in the drivers seat now. The monster computers are still going to be there, but the revolution in business is the small computer. I don't know if the "ideal" computer is a TRS-80, and Apple or an IBM or any of the other computers out there. Each one has something to recommend it and, usually, a bunch of things to cause you to shy away from it. All are popular and are supported by the independent software and magazine suppliers, the Apple and the IBM are well supported by the independent hardware suppliers while the TRS-80 is marginally supported with brand-x hardware.

Which computer do I recommend? You may have guessed that my answer is "It all depends!" What do you want to do and how fast do you want it done? What kind of image do you want? I can't recommend the Apple for all things because it's not an "all things to all people" computer. For the same reason I can't recommend the TRS-80 or the IBM or the Atari or the Commodore or the Flash-97.2 or any other computer. They each have their reason for existence, and I guess that the buying public will determine what they need.

I myself believe that as long as there's a market for a given computer the computer will continue to exist. If the manufacturers want to play games with the buying public they will be the ones to suffer in the long run. I find myself looking at this area with a completely different viewpoint than I did way back in 1977. When the industry first started I guessed that someday I would own a microcomputer (little did I guess that I would own four of them). My first love was a baby blue IMSAI 8080 computer. Unfortunately I don't know a flip-flop from a diode! I didn't want something that I had to put together (because it would either never be put together or else it wouldn't work).

I wanted a computer that you could plug into the wall, turn it on and have it work! Back then that was an unreasonable thing to ask for. After several years on mainframe computers it was what I wanted

and I was willing to wait. No, I'm not going to tell you what I got because that's not important. What is important is that I learned that some of what was good for the big computers was good for the little computers. More importantly, I learned that what was good for the little computers was *always* good for the large mainframes. I discovered that the development techniques that were being touted for the big computers worked very well for the little computers.

What really happened was that I began talking to other computer users and discovered that everybody in the microcomputer world (that I was exposed to) wanted to learn more about how to program correctly. They wanted to do this programming with minimal difficulty and with maximum results. Again these are trends that were becoming popular in the management circles of large corporations — get the most for the buck regardless of whether it was a buck spent for hardware, software, or people. Maximize everything became the rallying cry (especially maximize profits).

Seeing the demand (I was in management at the time) I began to see what was needed. Could the two worlds be merged? I used the microcomputer at work to do some planning — more managers got interested. But something else happened. We were looking for programmers and there was a critical shortage of good programmers. I had a guy come along that was experienced in the Apple computer but had no other background in data processing. Could I use him?

My management was skeptical but I went ahead and hired him (I always trusted my judgement on people I hired). we were very lucky. This individual turned out to be a super programmer and I never regretted hiring him. He has learned to apply the Apple tricks and techniques to the mainframe and it has benefited him. At the same time, I was teaching him about the mainframe developmental techniques which provided him with still more abilities to meet deadlines and schedules. The ideas that I've presented here do work. They work in the business world and the work in the hobby world.

Well, where do we go from here? I certainly hope that you'll take the time to play with the techniques that we've discussed here and try to develop your own programming skills so that you will be a better programmer. I'd certainly hate to think that you became a worse programmer as a result of reading this book.

"Ok, but what if I can't come up with an idea *right now* so that I can practice?"

As usual, that's a very good question. Let's answer it by saying that there's always a lot of things that can be done with the CHEKBOOK program. For example, you could find a way to add a routine that will read a file which contains a list of people or companies that are paid on a regular basis and have the program determine if the time has come to pay them. This should be around five working days before the payment is due. The program could then prompt the user to make the payment and to specify the amount or to override the payment (in case he's a little short that month).

There are lots of other ideas that can be implemented in this program. As we mentioned, the routines to process the service charges and the interest payments have been omitted and these need to be added. You could play around with them. Another nice enhancement would be to add a routine that would keep a name associated with the account number to make it more recognizable. It might be something like "personal," "business," "gambling" or whatever. This would make it easier to decide which account was which. You might even display the account number with the names below them any time the user must select an account number. The user could then specify the sequence number instead of the account number. Remember, the fewer keystrokes the user has to make the fewer chances he has to make a mistake.

As we pointed out, ideas are everywhere and as you play with this program you'll undoubtedly come up with ideas of your own. Evaluate them and, if they seem reasonable, add them to the program. Just remember to make sure that you've thought out the impact of the changes to minimize any problems that you might encounter. Once you've planned the changes, carefully implement them and see how the work out. It's possible that you may decide they weren't as beneficial as you'd thought they might be so you'll want to back the changes out. Or perhaps they worked out better than you thought they would. Great!

There's so much that you can do that I hesitate to discuss it further. One point that I would like to make is that you are the only one that can really make the techniques in this book work for you. If you don't want to use them — don't. But do give them a try, they're a little difficult to get the first time around (it took me several passes at using them before I became comfortable with them). If you will keep at it the skills will become almost second nature and you'll experience minimal difficulty in the future.

Before we go, let's talk about an area that's been skipped in this book but is important enough to be mentioned. When you write a SYSTEM, that is, a collection of related programs, there is a need for a greater awareness of interfaces. Just as we were very careful in insuring the interfaces of modules in a program, so we must be careful with the programs in a system.

In a way, these programs become "system modules" in the same way as related lines of code became modules in a program. The only difference is one of scale. If we considered a module a routine, so we can consider a program a function. Each was treated as a "black box" in that it did a particular function. This is pretty abstract and it would probably help if we looked at an example.

Let's assume that we are writing a payroll system. This system must read a file from another system (let's call it the personnel system) which contains the records of all the employees in the company and their pay rates. We will match the information (social security number, employee number, name, etc.) with the corresponding information in the "hours worked" file. Once we've made a match we'll compute the salary, taxes, deductions, etc. This information will then be posted in the various accounting system files. We will need to write checks for the employees and also tell the federal government about the withholding of FICA (Social Security) taxes.

Let's assume that we will have four programs in the payroll system; (1) Employee File Update Program, (2) Accounting File Update Program, (3) Check Writing Program, and (4) Government Notification Program. We could see that using our original definitions (from way back in Section 1) these are "routines" in the sense that they can be expressed as a single idea. The problem here is, as we said, one of scale. We've simplified the statements to the point that there are actually multiple layers of code needed to make the "routines" work. They are validly considered as programs — system modules if you will.

This does bring up an interesting question — how do we determine when we've broken an idea down to the bottom level, that is, to the "routine" level as opposed to the program level? That question is, unfortunately, hard to answer. Perhaps the best answer that I could give is to take a look at the routine and see if there are any underlying steps that are not "primitive" or elementary steps. In other words, if any of the underlying steps could be a routine itself then we're not down to the bottom layer. Is that clear?

Again, a system is made up of programs. Each program has the same relationship to a system as the modules of a program do to that program. This is a very important point, and really needs to be understood. Most programmers have no difficulty in making sure that *files* are correctly defined between different programs, but even that obvious interface can sometimes be a problem. A more severe problem, however, is that there are often data dependent interfaces that are not passed. For example, what if there are no checks to be written? Can the payroll system handle that fact or will if have a problem?

As we've already mentioned, sometimes one system will interface with another system. Now the problem of scale becomes even larger. At this point the systems become modules to a super-system. This is usually as large as we would want to get in an analysis of this kind. This sort of interfacing is no more difficult than that which you've already faced in designing and developing your programs. Don't let it bother you (assuming that you're the sort of person that might be bothered by this sort of thing).

To try and place this whole discussion in perspective, we can take a look at a company. Each individual within a company has someone to whom he reports. At each level of the company as we go up there are increasing levels of responsibility with (hopefully) increasing levels of authority. Lines of code are the bottom layer of "data processing" and as we increase we go to modules (or subroutines) to programs to systems to super-systems to THE SYSTEM which is usually the all encompassing term used to describe the hardware on which the software runs.

Each layer is important, and just as with a company, the bottom layers are more critical to the success or failure of the organization than the higher layers. This means that an entire system can fail if a single line of code fails. Seem strange or unbelievable? Recently the United States was placed on a "Yellow Alert" because of a faulty line of code in a NORAD (North American RADar Defense) program.

There can never be enough emphasis placed upon the importance of properly defining and writing each line of code. If we never have a failure of the lines of code, if the interfaces between the modules are carefully though out and tested, if the programs are tested properly and carefully fitted into the systems there will be fewer problems. Those that remain will probably be in the realm of "pilot error." Places where the user or the operator made a mistake

— not a data entry error because we should catch those — but a system control error.

For example, how do you go about catching the fact that the user has hit the reset button? On some computers you could trap the interrupt caused by the reset but that's not the sort of thing that would be done at the "application" programmer level. That sort of thing is usually left to a type of programmer called a "systems programmer." Most microcomputer programmers eventually become systems programmers by learning machine or assembly language for their computers. At that level they can then trap reset buttons (maybe).

We must always remember the immortal words of Edsel T. Murphy, "If, under any given set of circumstances anything can go wrong, it invariably will!" This axiom has guided the lives of most of the programmers that I've ever met or heard about. We live in constant awareness of that axiom -- but we don't let it get us down. We continue to develop and refine our own skills (and we find hiding places where no one else can find us, especially our bosses).

So keep practicing your programming skills. And, if you didn't like this book you probably won't like "How to Write a Computer Program Volume II, Advanced Programming Concepts" which will be available in a few months. This book will take you into some of the neater areas of programming including taking a good look at sorting, random number generation, some game concepts as well as a good look at file management techniques. Keep an eye out for it at your favorite computer or book store.

BIBLIOGRAPHY

Barden, Jr., W., PROGRAMMING TECHNIQUES FOR LEVEL II BASIC, Radio Shack, 1980

Bauer, F. L., SOFTWARE ENGINEERING, Springer-Verlag, 1975

Couger, J. D. and McFadden, F. R., INTRODUCTION TO COMPUTER BASED SYSTEMS, Wiley, 1975

Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R., STRUCTUREDPROGRAMMING, Academic Press, 1972

Jones, C. B., SOFTWARE DEVELOPMENT: A RIGOROUS APPROACH, Prentice-Hall International, 1980

Kernighan, B. W., and Plauger, P. J., THE ELEMENTS OF PROGRAMMING STYLE, McGraw-Hill, 1978

————, SOFTWARE TOOLS, Addison-Wesley, 1976

Lien, D. A., THE BASIC HANDBOOK 2nd EDITION, Compusoft Publishing, 1981

Myers, G. J., THE ART OF SOFTWARE TESTING, Wiley-Interscience, 1979

Nevison, J. M., THE LITTLE BOOK OF BASIC STYLE, Addison-Wesley, 1978

Rosenfelder, L., BASIC FASTER AND BETTER, IJG, 1981

Yourdon, E., MANAGING THE STRUCTURED TECHNIQUES, Yourdon Press, 1979

Zelkowitz, M. V., Shaw, A. C., and Gannon, J. D., PRINCIPLES OF SOFTWARE ENGINEERING AND DESIGN, Prentice-Hall, 1979

INDEX

APPENDIX A
MODULAR PROGRAM LISTING

```
10 REM              CHECKBOOK PROGRAM
20 REM  This program will process multiple accounts (w
   ith the limit currently set to 5).
30 REM
40 CLEAR 4000                        'RESERVE STRING SPACE
50 DEFDBL A
60 MN=150                            'CHECKS PER MONTH
70 MA=5                              'MAX ACCOUNTS
80 DIM CR$(MN,MA),AB(MA),AN(MA)      'Chk Reg, Acct. Bal.,
   Acct #
90 DIM CN(MA),NU$(25),MO$(12)        'CHK NOS, AMOUNTS, MO
   NTHS
100 REM ------------------------------------------------
110 REM         OPEN FILES, PROCESS DATE
120 REM
130 CLS
140 T$="CHEKBOOK -- A CHECK ACCOUNTING PROGRAM"
150 PRINT TAB(32-(LEN(T$)/2))T$
160 PRINT : PRINT "SYSTEM INITIALIZATION IN PROCESS, P
    LEASE WAIT"
170 ON ERROR GOTO 250
180 OPEN "I",1,"CRF/DAT" : ON ERROR GOTO 0
190 FOR I=1 TO MA : INPUT #1,AN(I) : NEXT I
200 FOR I=1 TO MA : INPUT #1,AB(I) : NEXT I
210 FOR I=1 TO MA : INPUT #1,CN(I) : NEXT I
220 IF EOF(1) CLOSE : GOTO 300
230 INPUT #1,I,J : LINE INPUT #1,CR$(I,J)
240 GOTO 220
250 PRINT "ERROR, CHECK REGISTER FILE NOT FOUND. FILE
    WILL BE CREATED!"
260 PRINT "HIT ENTER TO CONTINUE"
270 A$=INKEY$ : IF A$="" GOTO 270 ELSE IF ASC(A$)<>13
    GOTO 270
280 RESUME 290
290 ON ERROR GOTO 0
300 DT$=LEFT$(TIME$,8) : GOSUB 2510
310 IF VAL(DT$)<>0 GOTO 340
320 LINE INPUT "ENTER TODAY'S DATE (MM/DD/YY): ";DT$
```

```
330 GOTO 310
340 REM ------------------------------------------------
350 REM  DISPLAY MENU, GET & VALIDATE OPTIONS
360 REM
370 CLS
380 PRINT STRING$(24,"-")" CHEKBOOK MENU "STRING$(25,"
    -")
390 PRINT
400 PRINT "      1 -- POST        POST A HAND WRITTEN CH
    ECK"
410 PRINT "      2 -- WRITE       WRITE A CHECK"
420 PRINT "      3 -- STATEMENT   GENERATE THE CHECK STA
    TEMENTS"
430 PRINT "      4 -- SETUP       SETUP A NEW ACCOUNT"
440 PRINT "      5 -- DEPOSIT     POST A DEPOSIT TO ACCO
    UNT"
450 PRINT "      6 -- PURGE       PURGE CURRENT MONTH'S
    CRF"
460 PRINT "      7 -- END         TERMINATE THE PROGRAM"
470 PRINT@ 182,LEFT$(TIME$,8);
480 PRINT@ 246,RIGHT$(TIME$,8);
490 PRINT@ 64,"SELECT OPTION ===> ";CHR$(14);
500 A$=INKEY$ : IF A$="" GOTO 500
510 PRINT A$;CHR$(15); : OP=VAL(A$)
520 IF OP>=1 AND OP<=7 GOTO 550
530 PRINT@ 50,"INVALID OPTION";
540 GOTO 490
550 ON OP GOSUB 940, 1240, 1300, 1860, 2070, 2810, 227
    0
560 GOTO 370
570 REM ------------------------------------------------
580 REM     COMMON CHECK DATA INPUT ROUTINE
590 REM
600 PRINT@ 128,CHR$(31)
610 PRINT@ 128,"";
620 CD$(1)="      ACCOUNT: "+STRING$(15,CHR$(95))
630 CD$(2)="      WRITTEN TO: "+STRING$(25,CHR$(95))
640 CD$(3)="      CHECK AMOUNT: $"+STRING$(7,CHR$(95))
650 CD$(4)="      NOTES: "+STRING$(25,CHR$(95))
660 DX$=STRING$(2,CHR$(95)) : DX$=DX$+"/"+DX$+"/"+DX$
670 CD$(5)="      DATE WRITTEN: "+DX$
680 CD$(6)="      CHECK NUMBER: "+STRING$(7,CHR$(95))
690 CN=-1
700 FOR I=1 TO 4 : PRINT CD$(I) : NEXT I
710 IF CC=1 PRINT CD$(5) : PRINT CD$(6)
720 PRINT@ 142,CHR$(14); : GOSUB 840 : AC=VAL(IX$)
730 PRINT@ 209,CHR$(14); : GOSUB 840 : WT$=IX$
740 PRINT@ 276,CHR$(14); : GOSUB 840 : AM=VAL(IX$)
750 PRINT@ 332,CHR$(14); : GOSUB 840 : NT$=IX$
760 IF CC<>1 GOTO 800
770 PRINT@ 403,CHR$(14); : GOSUB 840  : DW$=IX$
780 PRINT@ 467,CHR$(14); : GOSUB 840  : CN=VAL(IX$)
```

```
800 PRINT@ 512,"";  : INPUT "IS THIS CORRECT";AN$
810 IF AN$="NO" OR AN$="N" GOTO 600
820 IF AN$<>"YES" AND AN$<>"Y" GOTO 800
830 PRINT@ 512,CHR$(31);  : RETURN
840 IX$="" : A$=INKEY$
850 A$=INKEY$ : IF A$="" GOTO 850 ELSE IF ASC(A$)<>13
    PRINT A$;
860 IF ASC(A$)>31 THEN IX$=IX$+A$ : GOTO 850
870 IF ASC(A$)=8 THEN IF LEN(IX$)>0 THEN IX$=LEFT$(IX$
    ,LEN(IX$)-1) : GOTO 850
880 IF ASC(A$)=24 PRINT " ";  : FOR I=1 TO LEN(IX$) : P
    RINT CHR$(8);  : NEXT I : IX$="" : GOTO 850
890 IF ASC(A$)=13 PRINT CHR$(15);  : RETURN
900 GOTO 850
910 REM ------------------------------------------------
920 REM      POST A HAND WRITTEN CHECK HERE
930 REM
940 IF AN(1)=0 GOSUB 1860
950 CLS
960 PRINT STRING$(20,"-")" CHECK POSTING ROUTINE " STR
    ING$(21,"-")
970 CC=1 : GOSUB 600
980 FOR I=1 TO MA : IF AC=AN(I) GOTO 1020 ELSE NEXT I
990 PRINT "ERROR, ACCOUNT NOT FOUND. RE-ENTER."
1000 INPUT "ACCOUNT NUMBER";AC
1010 GOTO 980
1020 FOR J=1 TO MN : IF CR$(J,I)="" GOTO 1050 ELSE NEX
     T J
1030 PRINT "ERROR, CHECK REGISTER IS FULL. HIT ENTER T
     O CONTINUE."
1040 GOTO 1190
1050 CR$(J,I)=WT$+CHR$(255)+STR$(AM)+CHR$(255)+NT$+CHR
     $(255)
1060 IF DW$<>"" CR$(J,I)=CR$(J,I)+DW$ ELSE CR$(J,I)=CR
     $(J,I)+LEFT$(TIME$,8)
1070 IF CC<>0 GOTO 1130
1080 IF (AB(I)-AM)>=0 GOTO 1130 ELSE PRINT "BALANCE WI
     LL GO NEGATIVE."
1090 INPUT "WRITE CHECK ANYWAY";AN$
1100 IF AN$="YES" OR AN$="Y" GOTO 1130
1110 IF AN$="NO" OR AN$="N" CR$(J,I)="" : RETURN
1120 GOTO 1090
1130 AB(I)=AB(I)-AM
1140 IF CN=-1 THEN CN=CN(I) : CN(I)=CN(I)+1
1150 CR$(J,I)=STR$(CN)+CHR$(255)+CR$(J,I)
1160 IF AB(I)<0 PRINT "BALANCE IS NOW NEGATIVE."
1170 IF OP<>1 RETURN
1180 PRINT "ACCOUNT POSTED. HIT ENTER TO CONTINUE."
1190 A$=INKEY$ : IF A$="" GOTO 1190 ELSE IF ASC(A$)<>1
     3 GOTO 1190
1200 RETURN
1210 REM ------------------------------------------------
```

```
1220 REM          WRITE A CHECK ROUTINE
1230 REM
1240 IF AN(1)=0 GOSUB 1860
1250 CLS
1260 PRINT STRING$(20,"-")" CHECK WRITING ROUTINE "STR
     ING$(21,"-")
1270 CC=0 : GOSUB 600  : GOSUB 980
1280 IF AN$<>"N" AND AN$<>"NO" GOSUB 2550
1290 RETURN
1300 REM ------------------------------------------
1310 REM          PRINT CHECK REGISTER
1320 REM
1330 CLS
1340 PRINT STRING$(21,"-")" PRINT CHECK REGISTER "STRI
     NG$(21,"-")
1350 PRINT
1360 HD$="CHK #  TO WHOM WRITTEN    AMOUNT          NO
     TES                  DATE"
1370 F$="#####  %                          %$##,###.## %
     %  %       %"
1380 D$="       %                          %$##,###.## %
     %  %       %"
1390 INPUT "FOR ALL ACCOUNTS";AN$
1400 IF AN$="YES" OR AN$="Y" GOTO 1460
1410 IF AN$<>"NO" AND AN$<>"N" GOTO 1390
1420 INPUT "ACCOUNT";AC
1430 FOR I=1 TO MA : IF AN(I)=AC GOTO 1470 ELSE NEXT I
1440 PRINT "ERROR, ACCOUNT NOT FOUND, RE-ENTER."
1450 GOTO 1420
1460 I=1
1470 LPRINT "ACCOUNT NUMBER: "AN(I);TAB(59);
1480 LPRINT "REPORT DATE: ";LEFT$(TIME$,8)
1490 LPRINT " "
1500 LPRINT "CHECK REGISTER:"
1510 LPRINT " "
1520 LPRINT HD$
1530 LPRINT " "
1540 CA=0 : DP=0
1550 J=1
1560 IF CR$(J,I)="" GOTO 1720
1570 CR$=CR$(J,I)
1580 FOR K=1 TO 4
1590    L=INSTR(CR$,CHR$(255))
1600    PT$(K)=LEFT$(CR$,L-1)
1610    CR$=RIGHT$(CR$,LEN(CR$)-L)
1620 NEXT K
1630 CN=VAL(PT$(1))
1640 AM=VAL(PT$(3))
1650 IF CN>=0 THEN CA=CA+AM : GOTO 1690
1660 LPRINT USING D$;PT$(2),AM,PT$(4),CR$
1670 DP=DP+AM
```

```
1680 GOTO 1700
1690 LPRINT USING F$;CN,PT$(2),AM,PT$(4),CR$
1700 J=J+1
1710 GOTO 1560
1720 LPRINT " "
1730 LPRINT USING "BEGINNING BALANCE:      $##,###.##";
     AB(I)+CA-DP
1740 LPRINT USING "DEPOSITS:               $##,###.##";DP
1750 LPRINT USING "TOTAL CHECKS:           $##,###.##";CA
1760 LPRINT USING "CURRENT BALANCE:        $##,###.##";
     AB(I)
1770 IF AN$="NO" OR AN$="N" GOTO 1820
1780 I=I+1
1790 LPRINT " "
1800 LPRINT " "
1810 IF I<MA AND AN(I)<>0 GOTO 1470
1820 IF OP<>3 RETURN
1830 PRINT "HIT ENTER TO CONTINUE"
1840 A$=INKEY$ : IF A$="" GOTO 1840 ELSE IF ASC(A$)<>1
     3 GOTO 1840
1850 RETURN
1860 REM ------------------------------------------
1870 REM           SET-UP AN ACCOUNT
1880 REM
1890 CLS
1900 PRINT STRING$(20,"-")" ACCOUNT SET-UP ROUTINE "ST
     RING$(20,"-")
1910 PRINT
1920 IF AN(1)=0 PRINT "NO ACCOUNTS SET-UP YET."
1930 PRINT
1940 FOR I=1 TO MA : IF AN(I)=0 GOTO 1970 ELSE NEXT I
1950 PRINT "ERROR, ACCOUNT REGISTER FULL"
1960 RETURN
1970 INPUT "ACCOUNT NUMBER";AN(I)
1980 INPUT "STARTING ACCOUNT BALANCE";AB(I)
1990 INPUT "NEXT CHECK NUMBER FOR THIS ACCOUNT";CN(I)
2000 PRINT
2010 PRINT "ACCOUNT ESTABLISHED. HIT ENTER TO CONTINUE
     ."
2020 A$=INKEY$ : IF A$="" GOTO 2020 ELSE IF ASC(A$)<>1
     3 GOTO 2020
2030 RETURN
2040 REM ------------------------------------------
2050 REM             POST A DEPOSIT
2060 REM
2070 CLS
2080 PRINT STRING$(24,"-")" POST A DEPOSIT "
     STRING$(24,"-")
2090 PRINT
2100 INPUT "ACCOUNT NUMBER";AC
2110 FOR I=1 TO MA : IF AN(I)=AC GOTO 2140 ELSE NEXT I
```

```
2120 PRINT "ERROR, ACCOUNT NUMBER NOT FOUND. RE-ENTER"
2130 GOTO 2100
2140 INPUT "DEPOSIT AMOUNT";DA
2150 FOR J=1 TO MN : IF CR$(J,I)="" GOTO 2190 ELSE NEX
     T J
2160 PRINT "ERROR, CHECK REGISTER FULL. HIT ENTER TO C
     ONTINUE"
2170 A$=INKEY$ : IF A$="" GOTO 2170 ELSE IF ASC(A$)<>1
     3 GOTO 2170
2180 GOTO 2230
2190 CR$(J,I)="-1"+CHR$(255)+" "+CHR$(255)+STR$(DA)+CH
     R$(255)+"DEPOSIT"+CHR$(255)+LEFT$(TIME$,8)
2200 AB(I)=AB(I)+DA
2210 PRINT "DEPOSIT POSTED. HIT ENTER TO CONTINUE."
2220 A$=INKEY$ : IF A$="" GOTO 2220 ELSE IF ASC(A$)<>1
     3 GOTO 2220
2230 RETURN
2240 REM ------------------------------------------
2250 REM         TERMINATION PROCESSING
2260 REM
2270 CLS
2280 PRINT STRING$(21,"-")" TERMINATION PROCESSING "ST
     RING$(22,"-")
2290 PRINT
2300 PRINT "WRITING CHECK REGISTER FILE."
2310 OPEN "O",1,"CRF/DAT"
2320 FOR I=1 TO MA : PRINT #1,AN(I) : NEXT I
2330 FOR I=1 TO MA : PRINT #1,AB(I) : NEXT I
2340 FOR I=1 TO MA : PRINT #1,CN(I) : NEXT I
2350 FOR I=1 TO MA
2360    FOR J=1 TO MN
2370        IF CR$(J,I)="" GOTO 2400
2380        PRINT #1,J,I,CR$(J,I)
2390    NEXT J
2400 NEXT I
2410 CLOSE
2420 PRINT "PROGRAM ENDED."
2430 END
2440 REM ------------------------------------------
2450 REM      FORMAT & PRINT CHECK (DATA)
2460 REM
2470 DATA ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,NINE
     ,TEN
2480 DATA ELEVEN,TWELVE,THIRTEEN,FOURTEEN,FIFTEEN,TWEN
     TY,THIRTY
2490 DATA FOURTY,FIFTY,SIXTY,SEVENTY,EIGHTY,NINETY
2500 DATA JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,
     DEC
2510 FOR I=1 TO 23 : READ NU$(I) : NEXT I : FOR I=1 TO
     12 : READ MO$(I) : NEXT I : RETURN
2520 REM ------------------------------------------
```

```
2530 REM          FORMAT AND PRINT CHECK
2540 REM
2550 WD$=""
2560 A1=INT(AM/1000)
2570 A2=INT((AM-A1*1000)/100)
2580 A3=INT(AM-(A1*1000+A2*100))
2590 A4=AM-INT(AM)
2600 IF A1<>0 THEN WD$=NU$(A1)+" THOUSAND "
2610 IF A2<>0 THEN WD$=WD$+NU$(A2)+" HUNDRED "
2620 IF A3=0 GOTO 2700
2630 IF A3<16 THEN WD$=WD$+NU$(A3) : GOTO 2700
2640 IF A3<20 GOTO 2690
2650 A5=INT(A3/10) : A3=A3-10*A5
2660 IF A3<>0 GOTO 2680
2670 WD$=WD$+NU$(A5+14) : GOTO 2700
2680 WD$=WD$+NU$(A5+14)+"-"+NU$(A3) : GOTO 2700
2690 WD$=WD$+NU$(A3-10)+"TEEN"
2700 WD$=WD$+" AND"+STR$(INT(A4*100))+"/100's"
2710 LPRINT TAB(41);MO$(VAL(TIME$));"";MID$(TIME$,4,2)
     ;",";TAB(56);MID$(TIME$,7,2)
2720 LPRINT " "
2740 LPRINT TAB(10);WT$;TAB(58); : LPRINT USING "**,##
     #.##";AM
2750 LPRINT " "
2760 LPRINT WD$
2770 LPRINT " "
2780 LPRINT " " : LPRINT " "
2790 LPRINT TAB(5);NT$
2800 RETURN
2810 REM ------------------------------------------
2820 REM           PURGE CURRENT CRF
2830 REM
2840 CLS
2850 PRINT STRING$(22,"-")" PURGE CURRENT CRF "STRING$
     (23,"-")
2860 GOSUB 1360
2870 IF AN$="YES" OR AN$="Y" GOTO 2890
2880 FOR I=1 TO MA : IF AN(I)=AC GOTO 2900 ELSE NEXT I
2890 FOR I=1 TO MA
2900    FOR J=1 TO NM
2910        CR$(J,I)=""
2920    NEXT J
2930    IF AN$="NO" OR AN$="N" GOTO 2950
2940 NEXT I
2950 PRINT "CRF PURGED, HIT ENTER TO CONTINUE."
2960 A$=INKEY$ : IF A$="" GOTO 2960 ELSE IF ASC(A$)<>1
     3 GOTO 2960
2970 RETURN
```

APPENDIX B
STRUCTURED PROGRAM LISTING

```
10 REM  CHECKBOOK PROGRAM (STRUCTURED VERSION)
20 REM  This program will process multiple accounts (w
   ith the limit currently set to 5).
30 REM
40 CLEAR 4000                        'CLEAR STRING SPACE
50 REM ------------------------------------------------
60 REM               MAIN LINE CODE
70 REM
80 GOSUB 120                         'INITIALIZATION
90 GOSUB 590                         'MENU
100 ON OP GOSUB 800   , 1040 , 1250 , 1790 , 1980 , 217
    0 , 2350
110 GOTO 90
120 REM ------------------------------------------------
130 REM              INITIALIZATION
140 REM
150 CLS
160 PRINT TAB(13)"CHEKBOOK -- A CHECK ACCOUNTING PROGR
    AM"
170 PRINT
180 PRINT "SYSTEM INITIALIZATION IN PROGRESS, PLEASE W
    AIT."
190 DEFDBL A                         'FOR LONG ACCOUNT NOS
    .
200 MN=150                           'MAX CHECKS PER MONTH
210 MA=5                             'MAX ACCOUNTS
220 DIM CR$(MN,MA),AB(MA),AN(MA)  'CHK REG, ACCT BAL, A
    CCT #
230 DIM CN(MA),NU$(23),MO$(12)    'CHK NOS, AMOUNTS, MO
    NTHS
240 REM ------------------------------------------------
250 REM       INITIALIZE VARIOUS VARIABLES
260 REM
270 HD$="CHK #  TO WHOM WRITTEN      AMOUNT          NOT
    ES                  DATE"
280 F$="#####  %                          %$###,###.## % %
      %       %"
290 D$="       %                          %$###,###.## % %
      %       %"
300 CD$(1)="      ACCOUNT: "+STRING$(15,CHR$(95))
310 CD$(2)="      WRITTEN TO: "+STRING$(25,CHR$(95))
```

```
320 CD$(3)="      CHECK AMOUNT: $"+STRING$(7,CHR$(95))
330 CD$(4)="      NOTES: "+STRING$(25,CHR$(95))
340 DX$=STRING$(2,CHR$(95)) : DX$=DX$+"/"+DX$+"/"+DX$
350 CD$(5)="      DATE WRITTEN: "+DX$
360 CD$(6)="      CHECK NUMBER: "+STRING$(7,CHR$(95))
370 ON ERROR GOTO 450
380 OPEN "I",1,"CRF/DAT" : ON ERROR GOTO 0
390 FOR I=1 TO MA : INPUT #1,AN(I) : NEXT I
400 FOR I=1 TO MA : INPUT #1,AB(I) : NEXT I
410 FOR I-1 TO MA : INPUT #1,CN(I) : NEXT I
420 IF EOF(1) CLOSE : GOTO 500
430 INPUT #1,I,J,CR$(I,J)
440 GOTO 420
450 PRINT "ERROR, CHECK REGISTER FILE NOT FOUND. WILL
    BE CREATED!"
460 PRINT "HIT ENTER TO CONTINUE."
470 A$=INKEY$ : IF A$="" GOTO 470 ELSE IF ASC(A$)<>13
    GOTO 470
480 RESUME 490
490 ON ERROR GOTO 0
500 D$=LEFT$(TIME$,8)
510 FOR I=1 TO 23 : READ NU$(I) : NEXT I
520 FOR I=1 TO 12 : READ MO$(I) : NEXT I : D$=LEFT$(TI
    ME$,8)
530 IF VAL(D$)=0 LINE INPUT "ENTER TODAY'S DATE (MM/DD
    /YY): ";D$
540 RETURN
550 DATA ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,NINE,
    TEN
560 DATA ELEVEN,TWELVE,THIRTEEN,FOURTEEN,FIFTEEN,TWENT
    Y,THIRTY
570 DATA FOURTY,FIFTY,SIXTY,SEVENTY,EIGHTY,NINETY,JAN,
    FEB,MAR
580 DATA APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC
590 REM ------------------------------------------------
600 REM                MENU ROUTINE
610 REM
620 CLS
630 PRINT STRING$(24,"-")" CHEKBOOK MENU "STRING$(25"-
    ")
640 PRINT
650 PRINT "      1 -- POST POST A HAND WRITTEN CHECK"
660 PRINT "      2 -- WRITE      WRITE A CHECK"
670 PRINT "      3 -- STATEMENT GENERATE THE CHECK STAT
    EMENTS"
680 PRINT "      4 -- SETUP      SETUP A NEW ACCOUNT"
690 PRINT "      5 -- DEPOSIT POST A DEPOSIT TO ACCOUNT
    "
700 PRINT "      6 -- PURGE PURGE CURRENT MONTH'S CRF"
710 PRINT@ "      7 -- END        TERMINATE THE PROGRAM"
720 PRINT@ 182,LEFT$(TIME$,8);
730 PRINT@ 246,RIGHT$(TIME$,8);
```

```
740 PRINT@ 64,"SELECT OPTION ===>";CHR$(14);
750 A$=INKEY$ : IF A$="" GOTO 750
760 PRINT A$;CHR$(15); : OP=VAL(A$)
770 IF OP>=1 AND OP <=7 RETURN
780 PRINT@ 50,"INVALID OPTION";
790 GOTO 740
800 REM ------------------------------------------------
810 REM        POST A HAND WRITTEN CHECK
820 REM
830 IF AN(1)=0 GOSUB 1790            'SET UP AN ACCOUNT
840 CLS
850 PRINT STRING$(20,"-")" CHECK POSTING ROUTINE "STRI
    NG$(21,"-")
860 GOSUB 2550                      'COMMON INPUT ROUTINE
870 FOR I=1 TO MA : IF AC=AN(I) GOTO 910  ELSE NEXT I
880 PRINT "ERROR, ACCOUNT NOT FOUND, RE-ENTER."
890 INPUT "ACCOUNT NUMBER";AC
900 GOTO 870
910 FOR J=1 TO MN : IF CR$(J,I)="" GOTO 940 ELSE NEXT
    J
920 PRINT "ERROR, CHECK REGISTER IS FULL, HIT ENTER TO
    CONTINUE."
930 GOTO 1020
940 CR$(J,I)=WT$+CHR$(255)+STR$(AM)+CHR$(255)+NT$+CHR$
    (255)
950 IF DW$<>"" CR$(J,I)=CR$(J,I)+DW$ ELSE CR$(J,I)=CR$
    (J,I)+D$
960 IF OP=2 RETURN                   'CALLED BY "WRITE"
970 AB(I)=AB(I)-AM
980 IF CN=-1 THEN CN=CN(I) : CN(I)=CN(I)+1
990 CR$(J,I)=STR$(CN)+CHR$(255)+CR$(J,I)
1000 IF AB(I)<0 PRINT "BALANCE IS NOW NEGATIVE."
1010 PRINT "ACCOUNT POSTED, HIT ENTER TO CONTINUE."
1020 A$=INKEY$ : IF A$="" GOTO 1020 ELSE IF ASC(A$)<>1
     3 GOTO 1020
1030 RETURN
1040 REM ------------------------------------------------
1050 REM         WRITE A CHECK ROUTINE
1060 REM
1070 IF AN(1)=0 GOSUB 1790 'SET UP AN ACCOUNT
1080 CLS
1090 PRINT STRING$(20,"-")" CHECK WRITING ROUTINE "STR
     ING$(21,"-")
1100 GOSUB 2550 'COMMON INPUT ROUTINE
1110 GOSUB 870 'USE POST ROUTINE CODE
1120 IF (AB(I)-AM)>=0 GOTO 1170 ELSE PRINT "BALANCE WI
     LL GO NEGATIVE."
1130 INPUT "WRITE CHECK ANYWAY";AN$
1140 IF AN$="Y" OR AN$="YES" GOTO 1170
1150 IF AN$="N" OR AN$="NO" GOTO 1220
1160 GOTO 1130
1170 AB(I)=AB(I)-AM                  'POST ACCOUNT
```

```
1180 CN=CN(I) : CN(I)=CN(I)+1
1190 CR$(J,I)=STR$(CN)+CHR$(255)+CR$(J,I)
1200 GOSUB 2840 'FORMAT & PRINT CHECK
1210 PRINT "CHECK WRITTEN, ";
1220 PRINT "HIT ENTER TO CONTINUE."
1230 A$=INKEY$ : IF A$="" GOTO 1230 ELSE IF ASC($)<>13
     GOTO  1230
1240 RETURN
1250 REM ------------------------------------------
1260 REM          PRINT CHECK REGISTER
1270 REM
1280 CLS
1290 PRINT STRING$(21,"-")" PRINT CHECK REGISTER "STRI
     NG$(21,"-")
1300 PRINT
1310 INPUT "FOR ALL ACCOUNTS"__$
1320 IF AN$="YES" OR AN$="Y" GOTO 1380
1330 IF AN$<>"NO" AND AN$<>"N" GOTO 1310
1340 INPUT "ACCOUNT";AC
1350 FOR I=1 TO MA : IF AN (I)=AC GOTO 1390 ELSE NEXT
     I
1360 PRINT "ERROR, ACCOUNT NOT FOUND. RE-ENTER."
1370 GOTO 1340
1380 I=1
1390 LPRINT "ACCOUNT NUMBER: "AN(I)
1400 LPRINT "REPORT DATE: ";D$
1410 LPRINT " "
1420 LPRINT "CHECK REGISTER:"
1430 LPRINT " "
1440 LPRINT HD$
1450 LPRINT " "
1460 CA=0 : DP=0
1470 J=1
1480 IF CR$(J,I)="" GOTO 1640
1490 CR$=CR$(J,I)
1500 FOR K=1 TO 4
1510    L=INSTR(CR$,CHR$(255))
1520    PT$(K)=LEFT$(CR$,L-1)
1530    CR$=RIGHT$(CR$,LEN(CR$)-L)
1540 NEXT K
1550 CN=VAL(PT$(1))
1560 AM=VAL(PT$(3))
1570 IF CN>=0 THEN CA=CA+AM : GOTO 1610
1580 DP=DP+AM
1590 LPRINT USING D1$;PT$(2),AM,PT$(4),CR$
1600 GOTO 1620
1610 LPRINT USING F1$;CN,PT$(2),AM,PT$(4),CR$
1620 J=J+1
1630 GOTO 1480
1640 LPRINT " "
1650 LPRINT USING "BEGINNING BALANCE: $##,###.##";AB
     (I)+CA-DP
```

```
1660 LPRINT USING "DEPOSITS:           $##,###.##";DP
1670 LPRINT USING "TOTAL CHECKS:       $##,###.##";CA
1680 LPRINT USING "CURRENT BALANCE:    $##,###.##";AB
     (I)
1690 IF AN$="NO" OR AN$="N" GOTO 1750
1700 I=I+1
1710 LPRINT " "
1720 LPRINT " "
1730 IF I<MA AND AN(I)<>0 GOTO 1390
1740 IF OP<>3 RETURN
1750 PRINT "STATEMENT"; : IF AN$<>"NO" AND AN$<>"N" PR
     INT "S";
1760 PRINT " PRINTED. HIT ENTER TO CONTINUE."
1770 A$=INKEY$ : IF A$="" GOTO 1770 ELSE IF ASC(A$)<>1
     3 GOTO 1770
1780 RETURN
1790 REM ------------------------------------------
1800 REM           SET-UP AN ACCOUNT
1810 REM
1820 CLS
1830 PRINT STRING$(20,"-")" ACCOUNT SET-UP ROUTINE "ST
     RING$(20,"-")
1840 PRINT
1850 IF AN(1)=0 PRINT "NO ACCOUNTS SET-UP YET."
1860 PRINT
1870 FOR I=1 TO MA : IF AN(I)=0 GOTO 1900 ELSE NEXT I
1880 PRINT "ERROR, ACCOUNT REGISTER FULL. ";
1890 GOTO 1950
1900 INPUT "ACCOUNT NUMBER";AN(I)
1910 INPUT "STARTING BALANCE";AB(I)
1920 INPUT "NEXT CHECK NUMBER FOR THIS ACCOUNT";CN(I)
1930 PRINT : IF OP<>4 RETURN
1940 PRINT "ACCOUNT ESTABLISHED, ";
1950 PRINT" HIT ENTER TO CONTINUE."
1960 A$=INKEY$ : IF A$="" GOTO 1960 ELSE IF ASC(A$)<>1
     3 GOTO 1960
1970 RETURN
1980 REM ------------------------------------------
1990 REM             POST A DEPOSIT
2000 REM
2010 CLS
2020 PRINT STRING$(24,"-")" POST A DEPOSIT "STRING$(24
     (24,"-")
2030 PRINT
2040 INPUT "ACCOUNT NUMBER";AC
2050 FOR I=1 TO MA : IF AN(I)=AC GOTO 2080 ELSE NEXT I
2060 PRINT "ERROR, ACCOUNT NUMBER NOT FOUND. RE-ENTER."
2070 GOTO 2040
2080 INPUT "DEPOSIT AMOUNT";DA
2090 FOR J=1 TO MA : IF CR$(J,I)="" GOTO 2120 ELSE NEX
     T J
```

```
2100 PRINT "ERROR, CHECK REGISTER FULL. HIT ENTER TO C
     ONTINUE"
2110 GOTO 2150
2120 CR$(J,I)="-1"+CHR$(255)+" "+CHR$(255)+STR$(DA)+CH
     R$(255)+"DEPOSIT"+CHR$(255)+D$
2130 AB(I)=AB(I)+DA
2140 PRINT "DEPOSIT POSTED. HIT ENTER TO CONTINUE."
2150 A$=INKEY$ : IF A$="" GOTO 2150 ELSE IF ASC(A$)<>1
     3 GOTO 2150
2160 RETURN
2170 REM ------------------------------------------
2180 REM            PURGE CURRENT CRF
2190 REM
2200 CLS
2210 PRINT STRING$(22,"-")" PURGE CURRENT CRF "STRING$
     (23,"-")
2220 PRINT
2230 GOSUB 1310              'PRINT FINAL STATEMENTS(S)
2240 IF AN$="YES" OR AN$="Y" GOTO 2260
2250 FOR I=1 TO MA : IF AC=AN(I) GOTO 2270 ELSE NEXT I
2260 FOR I=1 TO MA
2270    FOR J=1 TO MN
2280        CR$(J,I)=""
2290    NEXT J
2300    IF AN$="NO" OR AN$="N" GOTO 2320
2310 NEXT I
2320 PRINT "CRF PURGED. HIT ENTER TO CONTINUE."
2330 A$=INKEY$ : IF A$="" GOTO 2330 ELSE IF ASC (A$)<>
     D13 GOTO 2330
2340 RETURN
2350 REM ------------------------------------------
2360 REM          TERMINATION ROUTINE
2370 REM
2380 CLS
2390 PRINT STRING$(21,"-")" TERMINATION ROUTINE "STRIN
     G$(22,"-")
2400 PRINT
2410 PRINT "WRITING CHECK REGISTER FILE."
2420 OPEN "O",1,"CRF/DAT"
2430 FOR I=1 TO MA : PRINT #1,AN(I) : NEXT I
2440 FOR I=1 TO MA : PRINT #1,AB(I) : NEXT I
2450 FOR I=1 TO MA : PRINT #1,CN(I) : NEXT I
2460 FOR I=1 TO MA
2470    FOR J=1 TO MN
2480        IF CR$(J,I)="" GOTO 2510
2490        PRINT #1,J,I,CR$(J,I)
2500    NEXT J
2510 NEXT I
2520 CLOSE
2530 PRINT "PROGRAM ENDED."
2540 END
2550 REM ------------------------------------------
```

```
2560 REM          COMMON INPUT ROUTINE
2570 REM
2580 PRINT@ 128,CHR$(31)       'CLEAR TO END OF SCREEN
2590 PRINT@ 128,"";
2600 CN=-1
2610 FOR I=1 TO 4 : PRINT CD$(I) : NEXT I
2620 IF OP=1 PRINT CD$(5) : PRINT CD$(6)
2630 PRINT@ 142,CHR$(14); : GOSUB 2780 : AC=VAL(IX$)
2640 PRINT@ 209,CHR$(14); : GOSUB 2780 : WT$=IX$
2650 PRINT@ 276,CHR$(14); : GOSUB 2780 : AM=VAL(IX$)
2660 PRINT@ 332,CHR$(14); : GOSUB 2780 : NT$=IX$
2670 IF OP<>1 GOTO 2710
2680 PRINT@ 403,CHR$(14); : GOSUB 2780 : DW$=IX$
2690 PRINT@ 467,CHR$(14); : GOSUB 2780 : CN=VAL(IX$)
2700 IF CN<1 AND OP=2 GGTO 2690
2710 PRINT@ 512,"";  : INPUT "IS THIS CORRECT";AN$
2720 IF AN$="NO" OR AN$="N" GOTO 2580
2730 IF AN$<>"YES" AND AN$<>"Y" GOTO 2710
2740 PRINT@ 512,CHR$(31); : RETURN
2750 REM ------------------------------------------
2760 REM    SPECIAL CHARACTER-BY-CHARACTER
                  INPUT ROUTINE
2770 REM
2780 IX$="" : A$=INKEY$
2790 A$=INKEY$ : IF A$="" GOTO 2790 ELSE IF ASC(A$)<>1
     3 PRINT A$;
2800 IF ASC(A$)>31 THEN IX$=IX$+A$ : GOTO 2790
2810 IF ASC(A$)=8 THEN IF LEN(IX$)>0 THEN IX$=LEFT$(IX
     $,LEN(IX$)-1) : GOTO 2790
2820 IF ASC(A$)=24 PRINT " "; : FOR I=1 TO LEN(IX$) :
     PRINT CHR$(8); : NEXT I : GOTO 2780
2830 IF ASC(A$)=13 PRINT CHR$(15); : RETURN
2840 REM ------------------------------------------
2850 REM          FORMAT & PRINT CHECK
2860 REM
2870 WD$=""
2880 A1=INT(AM/1000)
2890 A2=INT((AM-A1*1000)/100)
2900 A3=INT(AM-(A1*1000+A2*100))
2910 A4=AM-INT(AM)
2920 IF A1<>0 THEN WD$=NU$(A1)+" THOUSAND "
2930 IF A2<>0 THEN WD$=WD$+NU$(A2)+" HUNDRED "
2940 IF A3=0 GOTO  3020
2950 IF A3<16 THEN WD$=WD$+NU$(A3) : GOTO 3020
2960 IF A3<20 GOTO 3010
2970 A5=INT(A3/10) : A3=A3-10*A5
2980 IF A3<>0 GOTO 3000
2990 WD$=WD$+NU$(A5+14): GOTO 3020
3000 WD$=WD$+NU$(A5+14)+"-"+NU$(A3) : GOTO 3020
3010 WD$=WD$+NU$(A3-10)+"TEEN"
3020 WD$=WD$+" AND"+STR$(INT(A4*100))+"/100's"
3030 LPRINT TAB(41)MO$(VAL(D$));", ";MID$(D$,4,2);TAB(
```

```
    56);RIGHT$(D$,2)
3040 LPRINT " "
3050 LPRINT TAB(10);WT$;TAB(58); : LPRINT USING "**,**
    #,##";AM
3060 LPRINT " "
3070 LPRINT WD$
3080 LPRINT " " : LPRINT " "
3090 LPRINT " " : LPRINT " "
3100 LPRINT NT$
3110 RETURN
```

Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia
Toll free (800) 336-0338

ISBN 08359-2992-2

9748 Cozycroft Ave., Chatsworth, CA 91311. (213) 709-1202